MODIFYING AFOTEC'S SOFTWARE
MAINTAINABILITY EVALUATION GUIDELINES

THESIS

Stephen K. Johnson, B.S.
Captain, USAF

AFIT/GCS/ENG/88D-10

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/88D-10

MODIFYING AFOTEC'S SOFTWARE MAINTAINABILITY EVALUATION GUIDELINES

THESIS

Stephen K. Johnson, B.S.
Captain, USAF

AFIT/GCS/ENG/88D-10

Approved for public release; distribution unlimited

AFIT/GCS/ENG/88D-10

MODIFYING AFOTEC'S SOFTWARE MAINTAINABILITY EVALUATION GUIDELINES

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Stephen K. Johnson, B.S.

Captain, USAF

December 1988

Preface

The purpose of this study was to develop a complexity metric or set of metrics that would be useful in measuring software maintainability. A set of interesting metrics was assembled from current literature, and a series of criteria was developed to measure how well each metric measures maintainability. Applying the criteria to the metrics, a pair of metrics that will best measure maintainability was determined.

Once the maintainability metrics were decided, rules for their implementation were given. A method to determine a threshold value was explained so that valid ranges of values could be recommended. A metric validation process was proposed to gather data that will reveal if the metrics actually reflect maintainability.

I would like to thank several people who have given me support and guidance throughout this thesis effort. I am very grateful to my advisor, Major James Howatt for his guidance in helping me narrow down my goals when I began research, his assistance throughout the development of this thesis, and his patience when I missed deadlines. I also wish to thank Captain Wade Shaw for explaining how important the metric threshold and validation analysis process is. A debt of gratitude is owed Captain David Umphress, whose editorial comments greatly enhanced the readability of this thesis. I would also like to thank my sponsor, Captain Mike McPherson, and Mr. Jim Baca of the Air Force Operational Test and Evaluation Center for their support and direction. Finally, I would like to thank all of my fellow students in the Computer Engineering section, who made the "AFIT Experience" unforgettable.

Stephen K. Johnson

Table of Contents

List of Figures

Abstract

The purpose of this ~~study~~ was to survey automatable software maintainability metrics for inclusion in the Air Force Operational Test and Evaluation Center's (AFOTEC's) software maintainability evaluations. This research was looking for metrics that would measure maintainability, could be automated, and would fit into existing guidelines. First, a set of software complexity metrics was investigated. Then, a set of criteria to determine if a complexity metric measures maintainability was developed. After comparing the metrics to the criteria, a subset of two metrics that met the criteria better than any other metrics was derived.

The software complexity metrics evaluated were placed into three categories: size metrics, structure metrics, and hybrid metrics. The structure metrics include both data structure and control structure metrics. The hybrid metrics include metrics blended from two of the other groups, such as a combination of size and structure metrics.

The metric selection criteria included three categories: general applicability criteria, control flow complexity criteria, and data flow complexity criteria. An assumption was made that the metric or combination of metrics that met the most of these criteria would best reflect software maintainability. A combination of a data structure metric, information flow, and a control structure metric, MEasurement Based on Weights (MEBOW), was determined to meet more criteria than any other metric or combination of metrics. This hybrid metric was suggested for AFOTEC use.

Further information explaining theoretical and empirical justification for the use of these metrics was given. A description of techniques to determine metric threshold values was discussed, along with a procedure for metric validation. Finally, a theme of the limitations inherent in measuring maintainability with automated metrics was elaborated.

MODIFYING AFOTEC'S SOFTWARE MAINTAINABILITY EVALUATION GUIDELINES

I. Introduction

Software metrics are tools to measure the intrinsic complexity of software systems in order to gauge the software design's "quality and effectiveness" (Prather, 1984:340). The quality of software should be measured to determine if it is both testable and maintainable (McCabe, 1983:3). These issues are important because testing requires a large amount of software development time, and software maintenance requires between 50 and 75 percent (Henry and Kafura, 1981:510) of the software life-cycle costs.

The Air Force Operational Test and Evaluation Center (AFOTEC) is responsible for testing software being developed for the Air Force. It uses software metrics to determine if software is maintainable. This thesis describes additional metrics that AFOTEC should use to measure maintainability. The use of these additional metrics will complement the current software evaluation guidelines.

Background

AFOTEC evaluates source code and documentation for the presence of seven maintainability characteristics. These characteristics are modularity, descriptiveness, consistency, simplicity, expandability, testability, and traceability. Each will be described later. Standardized questionnaires are filled out by software engineers who are "knowledgeable in software procedures, techniques, and maintenance, but need not have a detailed knowledge of the functional area of the program" (Peercy, 1981:343). The evaluators answer the questions within the questionnaire with respect to the software. Responses are analyzed and averaged to yield a maintainability rating.

Appraising software by this technique provides several advantages. This evaluation method can be used on any type of software, regardless of the implementation language. Weaknesses and strengths can be highlighted at any level, between subsystems within the software, or in comparison among systems. As both source code and documentation are considered, any discrepancies between how the specification says the software is constructed and how it is actually implemented can be discovered. AFOTEC's analysis of historical data suggests that their evaluation results correlate well with how difficult the software was to maintain. This implies that the current process does measure software maintainability.

This evaluation method has the disadvantage of being labor intensive. It requires that evaluators perform time-consuming activities such as counting the numbers of operands and operators in the source code. This means that it is expensive to assess software using this method. Because this evaluation is done manually, typically only about ten percent of the source code in large programs is examined. If this process were automated, all of the code could be measured, and the procedures that are shown to be more complex could later be evaluated in more detail using AFOTEC's current guidelines. The methodology used does not consider the overall software design, which is another drawback. Instead of judging design issues such as the connections between modules and how well the software has been modularized, the evaluation method looks at each module as a separate entity. If metrics that measure design complexity are used, the complexity of the inter-module data passing and the program calling structure can be considered. To eliminate these problems, additional software metrics should be used by AFOTEC to grade software, and they should be automated to reduce the evaluators' workload.

Summary of the AFOTEC Software Maintainability Guidelines

The following definitions describe what AFOTEC's guidelines are trying to measure. Then the criteria that used to measure maintainability are detailed. ANSI/IEEE Standard 729 (Schneidewind, 1987:303) states:

> Maintenance: Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.
>
> Maintainability: The ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements.

The AFOTEC pamphlet 800-2, Vol. 3 (referred to as the Vol. 3 from now on), "Software Maintainability - Evaluation Guide", standardized questionnaire assesses maintainability with respect to software source code and documentation. Quoting from the Vol. 3 itself, "These questionnaires [the Vol. 3] are designed to determine the presence or absence of certain desirable attributes in a given software product" (AFOTEC, 1988:1). These desirable attributes are the seven characteristics:

Modularity : "Software possesses the characteristic of modularity to the extent a logical partitioning of software into parts, components, and/or modules has occurred" (AFOTEC, 1988:5). The documentation is evaluated to determine if it is partitioned into separate parts or volumes that each has a distinct purpose. Similarly, source code is evaluated to determine the level of use of structured programming techniques.

Descriptiveness : "Software possesses the characteristic of descriptiveness to the extent that it contains information regarding its objectives, assumptions, inputs, processing, outputs, components, revision status, etc" (AFOTEC, 1988:5). This characteristic is used to measure how well described the software's design and operation is. Self-descriptive source language constructs and accompanying comments can facilitate efforts to understand program operation.

Consistency : "Software possesses the characteristic of consistency to the extent the software products correlate and contain uniform notation, terminology, and symbology" (AFOTEC, 1988:5). This characteristic is used to measure how well the software designers followed standards in creating documentation and how well coding conventions were followed. The use of a naming convention for global data and a standard indentation convention fall under this characteristic.

Simplicity : "Software possesses the characteristic of simplicity to the extent that it reflects the use of singular concepts and fundamental structures in organization, language, and implementation techniques" (AFOTEC, 1988:6). Simplicity is the overall guideline that size and control flow are measured against.

Expandability : "Software possesses the characteristic of expandability to the extent that a physical change to information, computational functions, data storage or execution time can be easily accomplished once the nature of what is to be changed is understood" (AFOTEC, 1988:6). Measuring expandability shows how much room for growth has been designed into a program in relation to its storage space, timing requirements, etc.

Testability : "Software possesses the characteristic of testability to the extent it contains aids which enhance testing" (AFOTEC, 1988:6). It is important that the software be instrumented for testing after modification, so that correct program execution can be verified and validated.

Traceability : "Software possesses the characteristic of traceability to the extent that information regarding all program elements, and their implementation, can be traced between all levels of lesser and greater detail" (AFOTEC, 1988:7). This characteristic measures how easily a maintainer can

trace the operation of a module to its documentation and can follow functions in the documentation to the modules that perform the functions.

These characteristics form the criteria for analysis of what makes software more maintainable. While these characteristics are not the only ones that can be measured to assess software maintainability (others include reliability, modifiability, etc.), they appear to be a representative sample of software quality characteristics (Boehm and others, 1980:229-231 and Peercy, 1981:343-344).

Problem

The AFOTEC evaluation guidelines are labor intensive and expensive to implement. These guidelines do not evaluate the overall software design. While the evaluation of different types of systems may require a different emphasis, no procedures exist to weigh the seven characteristics. Also, only a fraction of the delivered code is fully assessed.

Approach

Specific additional software maintainability metrics will be identified for incorporation into the Vol. 3. These metrics will measure aspects of maintainability that are presently not adequately covered. They will be automatable so that more labor will not be added to the evaluators' workload. Algorithms to develop a program to measure these metrics will be developed, although the actual measurement tool will not be built.

Assumptions

Two assumptions were made about the researched software metrics. First, the metrics must enhance the measurement of software maintainability. There are many different types of software metrics. Some measure other factors than

those related to maintainability. Second, the metrics must fit into the scope of the desirable attributes being measured. These desirable attributes were described in a previous section.

Scope

From the constraints explained in the previous section, the software metrics that will be suggested to AFOTEC will be limited to metrics that.

1. Will fit into the Vol. 3 process.
2. Can be automated.
3. AFOTEC can be convinced to use and acquire a tool to automate the calculation of these metrics.

A plan to validate these new metrics will be proposed. Algorithms to show how these metrics should measure source code will be developed.

Sequence of Presentation

Chapter Two presents a review of classic and newer software metrics. Chapter Three discusses criteria to determine which metrics measure maintainability. Chapter Four describes in detail which metrics will be suggested to AFOTEC and how these metrics should be incorporated within the existing guidelines of the Vol. 3, along with a method to validate the metrics' use. Chapter Five includes my conclusions and recommendations for further research.

II. Literature Review

Introduction

Research to date has not found metrics that specifically measure program maintainability; most metrics measure program complexity. Complexity can be defined as "a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software" (Conte and others, 1986:17). According to Harrison, "maintenance is most affected by program complexity" (Harrison and others, 1982:65). As program complexity greatly contributes to maintainability, I will use these complexity metrics to measure maintainability.

This chapter presents a review of classic and newer software metrics. Classic metrics are pioneering work such as as Lines of Code, Halstead's Software Science, and McCabe's cyclomatic complexity which have been extensively examined in the literature (Côté and others, 1988:121). The metrics are presented in three sections, as size metrics, as data and control structure metrics, and as composite or hybrid metrics. This list of metrics is not intended to be inclusive, but to show what factors of program complexity are measured and the metrics that attempt to measure these factors.

Size Metrics

Size metrics measure program size and reflect that the volume of information to be studied to understand the program contributes to its complexity (Harrison and others, 1982:66). Because the effort needed to develop a program largely depends on the amount of code written, size measures have been used to assess the amount of effort required. The program size is important for three reasons (Conte and others, 1986:32):

1. It is easy to compute after the program is completed.

2. It is the most important factor for many models of software development.

3. It is the basis of most productivity measures.

Lines of Code. The earliest and most familiar software size measure is the number of lines of source code (Levitin, 1986:314). This measure is labeled S and is measured in lines of code (LOC) or thousands of lines of code (KLOC). While this may seem to be a very simple and easily-calculated metric, much debate has centered around how LOC should be counted.

While this measure is natural for some languages such as various assembly languages and FORTRAN which have very close to a one-to-one correspondence between the number of statements and the lines of a program, newer languages that allow a more free format cannot be counted quite so easily. For example, Figure 1 shows two code fragments which are functionally equivalent, but have apparently different counts for LOC.

```
while V <> 0 do
  begin
    T := U mod V;
    U := V;
    V := T;
  end;
GCD := U;
```

```
while V <> 0 do begin
  T := U mod V; U := V; V := T
end;
GCD := U;
```

Figure 1. Differing Statement Counts for the Same Function (Levitin, 1986:315)

These examples have LOC counts of seven and four. As they are semantically the same program, the LOC count must be measuring the size of the program's representation, instead of the actual size of the program (Levitin,

1986:315). Another problem with the LOC count is that it is possible to pad the program with blank lines and comments to give artificially high LOC counts.

Many languages require descriptive non-executable statements, such as COBOL's ENVIRONMENT division or Pascal's Var section (Conte and others, 1986:35). Some researchers have suggested that since these are not executable statements they should not be counted in the LOC. Others have said that since understanding a program's data is critical to understanding the operation of the program, the variable declarations and program headers should be included in the LOC count. A simple solution to this problem is to have a consistent counting scheme and always use it to count LOC. An example of a counting strategy comes from Conte: "A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements" (Conte and others, 1986:35).

LOC is not a context sensitive software metric. As an example, twenty lines of conditional statements that control manipulating dynamic memory constructs will be inherently more complex than twenty lines of simple variable assignment statements. But with the LOC measure, each program will have the same count.

Halstead's Software Science (N). Dr. Maurice Halstead developed a measure of program size within his Software Science software metrics. This metric measures the number of operators and operands in a program (Levitin, 1986:316). Operators include arithmetic and logic symbols, functions, and delimiters such as + and −. Operands include variables, constants, and labels, and any other symbol that represents data.

From two basic quantities Halstead's program length metric, $\underline{N}$, is calculated (Conte and others, 1986:37):

$$\underline{N} = N1 + N2 \quad (1)$$

where

N1 = the total number of operators
N2 = the total number of operands

As with LOC, there is some difficulty determining what should be counted as an operator and what should be counted as an operand. In some languages, such as LISP, the difference between operators and operands is not clear. In a procedural language like Pascal or Ada, a function that is embedded in another function, such as "WRITE(COS(25))", can be considered both an operator and an operand. The COS[INE] function is an operator because it operates on the data that is input, but it is an operand also because its resulting value is used as data for the WRITE function.

Halstead originally stated in his counting rules that input/output statements and program declarations should not be counted. Also, the statement labels used as branching addresses for GOTO statements were not considered operands, but as an integral part of the GOTO's that branched to the label. Currently, research suggests Software Science counting rules should include counting the symbols in the declaration and input/output statements, as well as counting each distinct label as another operand (Shen and others, 1983:157).

Figure 2 shows Ramamurthy and Melton's example of counting operands and operators (Ramamurthy and Melton, 1986:309). The guideline of counting operators within the declaration statement is not followed in this example. The eleven operators are "BEGIN END", "readln", "()", ",", ";", ":=", "+", "*", "-", "writeln", and ".". These operators are used 23 times. The five operands are

```
PROGRAM1(input, output);

VAR
  a,b,c,d,m : integer;
BEGIN
  readln(a,b,c,d);
  a := a + b;
  b := a + c;
  c := b * d;
  m := a + b - c;
  writeln(m)
END.
```

Figure 2. Halstead's N Example
(Ramamurthy and Melton, 1986:309)

listed after the VAR statement. These operands are used 18 times. This gives values of 23 for N1, 18 for N2, and 41 for N.

While Halstead's overall theory of Software Science has been criticized as having no valid theoretical basis (Hamer and Frewin, 1982:198), the N measure has not been faulted as some other metrics have been. Shen states "there is a large amount of empirical evidence to suggest [N's] validity, although it appears to work best in the range of N between 2000 and 4000 for programs written in Fortran, Cobol, and PL/S" (Shen and others, 1983:163). But like LOC, this measure is not context sensitive in that it does not weight some operators as being inherently more complex than other operators.

Structure Metrics

This category investigates the system design structure, and constitutes the data relationships among system components and the control flow within system components. Some structure metrics have an advantage over size metrics because they can be applied early in the system lifecycle since they are based

on higher-level design features, not the actual source code (Kafura and Canning, 1985:379). Some control structure metrics can evaluate the complexity of a program's structure using its Program Design Language (PDL), which is available before the source code. Some data structure metrics can evaluate a program's complexity if the program's data flows are known before coding.

Data Structure Metrics. One factor that effects the complexity of a program is the amount of data the program uses, how it is used, and its configuration within the program.

Span. Span is a measure of the "number of statements between two successive references to the same variable" (Conte and others, 1986:56). A large span increases the difficulty of determining the value of the variable at any point. A large span could require a maintenance programmer to search through many lines of source code to understand a variable's usage (Harrison and others, 1982:67).

According to Harrison (Harrison and others, 1982:67), span is not supported by empirical evidence that it represents the complexity, and, therefore, maintainability of a program. But he does state that span is intuitively appealing, because a variable with a large span is inherently difficult to keep track of. Figure 3 shows Harrison's example of span between data references.

Information Flow. Previous metrics measure complexity within a single module. Because many programs contain more than one module, some way to measure the connections among modules is needed. Information flow is a method to measure the sharing of data among modules. The information flow metric captures properties of module connections that are more detailed than just "calling" relations. Information flow measures the amount of data that flows into a module and is modified by the module.

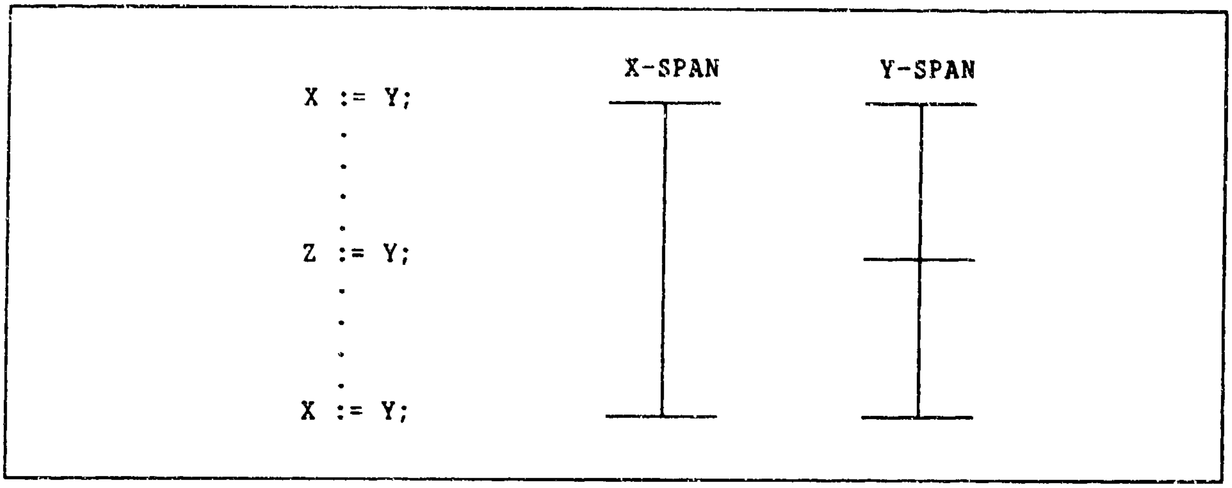

Figure 3. Span Example
(Harrison and others, 1982:66)

Harrison and Cook describe information flow as a "macrolevel" metric (Harrison and Cook, 1987:215). A macrolevel metric determines the interrelationships of the subprograms to each other in order to understand the behavior of the overall system. These metrics "concentrate on the communication links between subprograms--the more links, the more complex the macrolevel understanding" (Harrison and Cook, 1987:215). A potential problem with each link is that it may introduce "side effects" into other system subprograms (Harrison and Cook, 1987:215).

The information flow complexity value for a module is determined by two factors: the complexity of the module's code and the complexity of the connections to its environment (Henry and Kafura, 1981:513). The complexity of the connections is evaluated by the module's fan-in and fan-out. The fan-in is the number of local (parameter) data flows into the module and the number of global data structures that the module gets information from. The fan-out is

the number of data flows from the module and the number of global data structures that the module modifies.

It is interesting to note that while the complexity of the module's code is mentioned as a factor within information flow, it is disregarded in later calculations. Henry and Kafura state that "code length is only a weak factor in the complexity measure...this factor may be omitted without significant loss of accuracy" (Henry and Kafura, 1981:514). But other empirical validations of this metric (Kafura and Canning, and Harrison and Cook) use length as a factor in the information flow calculations. Kafura and Canning refer to the use of length (LOC) with information flow as "weighted information flow" and consider it a hybrid (Kafura and Canning, 1985: 380).

Henry and Kafura evaluated different formulas to calculate the complexity of the modules in the Unix kernel. The formulas included:

(length ** 2) (2)

(fan-in * fan-out) (3)

(fan-in * fan-out) ** 2 (4)

(length) * (fan-in * fan-out) ** 2 (5)

where length is the number of lines of text in a procedure, including embedded comments but not including the comments in the procedure's "header block" (Henry and Kafura, 1981:513).

They found "the connections of a procedure to its environment, namely (fan-in * fan-out) ** 2, is an extremely good indicator of complexity" (Henry and Kafura, 1981:516). Figure 4 shows an example information flow count using this formula. This is a correlation to the number of changes made to each module. They state that studies have shown a high correlation between program changes and error occurrences, which relates to maintainability (Henry and Kafura, 1981:515). Kafura also uses information flow to detect outliers in the

number of errors and amount of coding time required for large NASA Fortran projects, as further validation of the metric (Kafura and Canning, 1985:382). Outliers are those components that are more than one standard deviation above the mean for coding time required or in the number of errors they contain. Harrison states that his data "suggests that the metrics [information flow] work quite well in identifying 'extraordinary cases'" (Harrison and Cook, 1987:218).

```
procedure EXAMPLE1(Input1 : integer;
                   Input2 : integer;
               var Output1 : integer;
               var Output2 : integer);

  begin

    Output2 := 10;
    Global1 := Input1;
    Output1 := Input1 + Input2 + Global2;
    Global2 := Output1 * Output2;

  end;
```

Figure 4. Information Flow Example

Figure 4 shows an example of source code in a Pascal-like language. The procedure EXAMPLE1 has two local data flows and one global data flow into the procedure, for a total fan-in of three. This procedure has two local data flows and two global data flows out of the procedure, for a total fan-out of four. Note that the global variable Global2 is used as both an input and an output flow of information. The information flow complexity for this module = (3 * 4) ** 2, which is 144.

Coupling is "the degree of interdependence between two modules" (Page-Jones, 1980:101). Minimal coupling reflects that each module is as independent as possible from other modules. This indicates that a system has been partitioned appropriately (Page-Jones, 1980:101). The information flow metric can indicate the degree of coupling between procedures via the fan-in and fan-out terms. According to Henry and Kafura, this can reveal the existence of three types of problems for a procedure (Henry and Kafura, 1981:514). High fan-in or fan-out suggests that a procedure may perform more than one function, which is contrary to structured decomposition rules (Page-Jones, 1980:119). Related to this point is that information flow measurements may indicate a procedure that was inadequately refined and needs to be divided into two or more separate procedures. A procedure having high complexity may be a "stress point" that has a large amount of information traffic. Because of the large number of potential effects on the entire system, the procedure may be difficult to modify.

Henry and Kafura state that one of the benefits of information flow is that the data necessary to compute the metric is available during the design phase of software development (Henry and Kafura, 1981:511). This is a significant advantage over many of the metrics explained here, which cannot be measured until the source code has been delivered. Information flow is not related to the source language used, which means that it is widely applicable.

Control Structure Metrics. These metrics measure how easily understandable the control structures are in a program. These metrics measure the number of control transfers within a program, or how the control transfers are interrelated.

McCabe's Cyclomatic Complexity. Another classic metric is McCabe's cyclomatic complexity measure. Kearney notes that "McCabe considers the

program as a directed graph in which the edges are lines of control flow and the nodes are straight line segments of code. The cyclomatic number represents the number of linearly independent execution paths through the program" (Kearney, 1986:1045). The metric measures the number of basic paths through a program using graph theory to represent the paths instead of actually counting them, which may be impractical (McCabe, 1983:3).

To calculate a module's cyclomatic complexity, a directed graph "G" is generated, reflecting the control structure of the module. A node corresponds to a block of sequential code. An edge corresponds to a control transfer between nodes. The number of connected components is the number of distinct procedures, which is typically 1. The formula for calculating the cyclomatic complexity of a weakly connected flow graph is given in (McCabe, 1983:4) as:

$$v(G) = e - n + 2p \tag{6}$$

where

e = the number of edges
n = the number of nodes
p = the number of connected components

and v(G) is equal to the number of basic paths in the measured program. Figure 5 presents an example of code from Ramamurthy and Figure 6 shows its directed graph representation.

Construction of a directed graph can be time-consuming. Fortunately, Harlan D. Mills proved that the cyclomatic complexity of a structured program is one more than the number of decisions (McCabe, 1983:9). This means that v(G) can "be readily calculated by simply inspecting the program" (Myers, 1977:62) and automated program scanners have been built to calculate the complexity of programs.

From Figure 6 the number of nodes is n = 11. The number of edges connecting these nodes is e = 13, with an extra arc from the exit node to the entry node added to create a strongly connected graph. This extra arc adds one to the cyclomatic number, so the number of connected components p is used instead of 2p. The number of connected components is p = 1. Therefore, v(G) = 13 - 11 + 1 = 3. Note that this number can be easily calculated by inspecting the program. Adding one to the number of decisions (IF statements) equals 3.

```
PROGRAM2(input, output);

VAR
   a,b,c,d,m : integer;
BEGIN
  readln(a,b,c,d);
  IF a > b then
       IF b > c then
            m := a + b
       ELSE
            m := b + c
  ELSE
    m := c + d + a;
  writeln(m)
END.
```

Figure 5. Example Program
(Ramamurthy and Melton, 1986:309)

Empirical evidence supports the cyclomatic number as a complexity measure. Curtis stated that cyclomatic complexity is "related to the difficulty programmers experience in locating errors in code" (Curtis and others, 1980:307). Henry said the cyclomatic complexity metric is a "useful indicator of the occurrence of errors" (Henry and others, 1983:130).

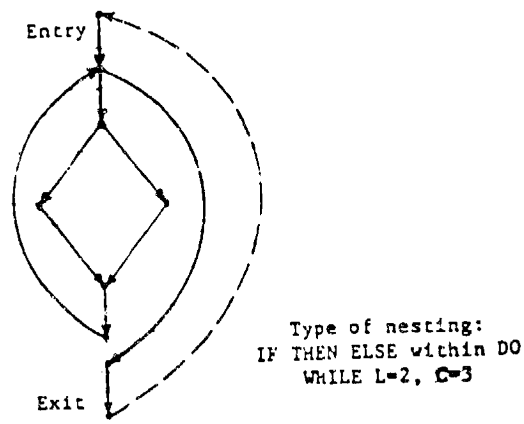

Figure 6. Example Directed Graph Representation of Figure 5 (Ramamurthy and Melton, 1986:310)

Shepperd, to the contrary, states that the high correlations obtained between cyclomatic complexity and errors is invalid because "the fundamental problem remains that without an explicit underlying model the empirical 'validation' is meaningless and there is no hypothesis to be refuted" (Shepperd, 1988:35). He points out that researchers have also tried to measure inter-module complexity with cyclomatic complexity, and that cyclomatic complexity can only measure intra-module complexity. McCabe suggests to measure the complexity of a program, the cyclomatic complexity of each module should be added to the number of modules, and an overall complexity score will be given. This does not take into account that an astute partitioning of a program into modules makes each smaller module's control flow easier to understand. Also, the data flow among modules is ignored completely.

A weakness of cyclomatic complexity is that it cannot measure the complexity of software that is due to size (Ramamurthy and Melton, 1986:310). A 10000-line program with only 8 decision points is intuitively complex, but its cyclomatic complexity is not high enough to attract attention, as McCabe states that 10 is a "reasonable" upper limit for cyclomatic complexity (McCabe, 1983:9). This example is somewhat far-fetched, but it illustrates the problem. Because cyclomatic complexity looks at a graphic representation of the program, and not the program itself, cyclomatic complexity will not be able to reveal a more structured version of the program because the same number of conditional statements will exist in each version (Woodward and others, 1983:103).

A problem that many researchers have had with cyclomatic complexity is that it does not take the nesting levels of branch statements into account (Myers, 1977:62). According to Harrison, "predicates with compound conditions are more complex than predicates with a single condition" (Harrison and others, 1982:70). Myers suggests calculating complexity as a range, with the lower bound as the number of decision statements plus one, and the upper bound as the number of individual conditions plus one (Myers, 1977:63). This modification of cyclomatic complexity apparently allows a finer distinction between programs with nested conditional statements, but no experiments have been published supporting this viewpoint.

<u>Knot Count</u>. Knot count was derived from two simple measures of complexity. In 1968, the <u>Communications of the ACM</u> published the now famous letter by Dr. Edsger W. Dijkstra entitled "Goto Statement Considered Harmful." In this letter, Dijkstra stated that the "quality of programmers is a decreasing function of the density of GOTO statements" (Woodward and others, 1983:101). This letter suggested that the number of GOTO statements in a program is a simple measure of unstructuredness. Discussing the theoretical basis of the

knot count metric, Woodward points out that in the book <u>Software Metrics</u> Gilb states that "logical complexity is a measure of the degree of decision making within a system and that the number of IF statements is a rough measure of this complexity" (Woodward and others, 1983:101).

Knot count measures the "relations between the physical locations of control transfers rather than simply their numbers" (Harrison and others, 1982:71). Knot count is a measure of program "unstructuredness", as it looks at the number of GOTO statements to count the number of crossing control transfers. These control transfer crossings are knots, and the greater the number of knots, the more complex the program is. Knots represent the "unstructuredness" of the source code text, but does not represent the program's underlying control flow (Howatt, 1988).

Woodward defines a knot as:

> If a jump from line a to line b is represented by the ordered pair of integers (a,b), then jump (p,q) gives rise to a "knot" or crossing point with respect to jump (a,b) if either
>
> 1) $\min(a,b) < \min(p,q) < \max(a,b)$ and $\max(p,q) > \max(a,b)$
>
> or
>
> 2) $\min(a,b) < \max(p,q) < \max(a,b)$ and $\min(p,q) < \min(a,b)$ [Woodward and others 1983:102]

An example from Woodward with nine knots is illustrated in Figure 7. An advantage that the knot count has over cyclomatic complexity is that a program that is unstructured and has a high knot count can be rewritten in a more structured fashion and have fewer knots. This is because the number of knots in a program depends on the order of the statements (Woodward and others, 1983:103). Harrison says the knot count is an interesting metric, but no research has applied it to the maintenance of programs (Harrison and others, 1982:71).

```
          CALL TPR
          IF (ZR) 500, 500, 100
     100  CALL TED
     150  IF (Z3) 200 200 550
     200  ZG = ZG + 1
          ZC = 0
          CALL TCO
     300  CALL TRA
          GOTO 2000
     500  CONTINUE
          Z3 = 1
          GOTO 150
     550  CONTINUE
          CALL TEC
          ZB = ZB + 1
          ZC = ZC + 1
          GOTO 300
    2000  RETURN
          END

    ╪ = KNOT
```

Figure 7. Knot Example
(Woodward and others, 1983:104)

<u>MEBOW (MEasure Based On Weights)</u>. MEBOW was developed as a control flow metric that measures complexity as well as the three metrics cyclomatic complexity, knot count, and Harrison's SCOPE ratio, but does not have their deficiencies (Jayaprakash and others, 1987:238). MEBOW is a modification of the cyclomatic complexity with the knot count added, and with different weights for control structures.

The developers of MEBOW state that the program complexity is best measured by control flow metrics, but that factors other than control flow metrics must also be considered (Jayaprakash and others, 1987:238). This is why they count knots and weigh different types of branch statements differently. Following Woodward's argument that knots create unstructured

programs, they state "the identification of knots helps in assigning higher control flow complexity to programs containing unstructured forms" (Jayaprakash and others, 1987:240).

MEBOW weighs a backward branch or knot higher than a forward branch or knot under the principle that while a backward branch may not necessarily lead to a loop, it does make a top-down reading of a program more difficult, and therefore more complex. Explicit branch statements such as a GOTO also have a higher weight than an implicit branch that is associated with a structured programming construct such as a FOR loop.

To determine the MEBOW value for a module, the sum of the weights of all branches and knots is calculated. Branches and knots are the only two basic programming elements that are assigned weights in MEBOW. This process is extended in the same manner as cyclomatic complexity is across modules, as the MEBOW value for more than one procedure within a program is the sum of each procedure's MEBOW value (Jayaprakash and others, 1987:241).

As with the knot count, there is no research that shows MEBOW effectively measures the maintainability of programs. Also, MEBOW's authors state that it can be used to measure inter-procedure complexity, but it ignores the data flow between procedures.

Composite (Hybrid) Metrics

A composite or hybrid metric is one that does not just measure a single factor to determine the complexity of software. As suggested by Kafura, Conte, and Hansen, different types of metrics measure significantly different aspects of software. For example, size metrics alone cannot reflect which of two 1000-line procedures is more complex. Control structure metrics cannot differentiate between one program that uses a global pointer structure and another program

that performs the same function that operates on an array passed as a parameter through a well-defined interface.

Since most metrics capture only one factor of complexity, it makes sense to use different metrics and to combine the results into a vector. Harrison, and Li and Cheung, all assert in different articles that using a hybrid metric to measure complexity is "the most sensible approach. Software complexity is caused by so many different factors that measuring only one of them cannot help but give unreliable results for a general case" (Harrison and others, 1982:78; Li and Cheung, 1987:708).

As empirical evidence that this type of composite metric does work, Kafura used a combination of the LOC and information flow metrics to determine procedures that had higher error and coding time rates in three large NASA Fortran projects. The composite metric determined the error and coding time outliers more often than any other code or structure metric he tested.

Two other composite metrics that have some data supporting their use are Ramamurthy and Melton's synthesis of Software Science metrics and the cyclomatic number, and Li and Cheung's NEW_1. The Software Science and cyclomatic complexity metric weighs the operator and operand count by the nesting level, so that an operator in a purely sequential program is not counted, while the same operator nested three levels deep would count as three operators. Ramamurthy and Melton have evidence that weighted length and effort detect differences in complexity between programs better than non-weighted length and effort do (Ramamurthy and Melton, 1986:312). NEW_1 is a composite of SCOPE, which is a control graph metric, and the Software Science effort metric. This metric is a combination of a graph metric with a size metric

in an attempt to receive the benefits of both types of metric (Li and Cheung, 1987:702).

When using two or more metrics, some difficulties interpreting data may arise. As an example, a and b are metrics. If with two procedures 1 and 2, a1 > a2 and b1 > b2, then procedure 1 is apparently more complex than procedure 2. But if the metrics are used to measure the same two procedures and results of a1 > a2 but b1 < b2 occur, it is not clear which procedure is more complex. According to Conte (Conte and others, 1986:80), this problem is why more researchers do not use composite metrics.

<u>Halstead's Software Science (E)</u>. Another Software Science metric is Effort "<u>E</u>", which is used to measure the number of "elementary mental discriminations" (Shen and others, 1983:156) that a programmer will have to make to produce the desired program. This is termed a hybrid because it calculated based on estimations of the number of "mental comparisons" needed to write a program of a certain length, and an estimation of the "program level", which represents a program written with minimum size (Conte and others, 1986:83). <u>E</u> can be approximated by (Conte and others, 1986:84):

$$\underline{E} = (\eta 1 * N2 * \underline{N} * \log 2\ \eta) / (2 * \eta 2) \tag{7}$$

where

$\eta 1$ = the number of unique operators
$\eta 2$ = the number of unique operands

$$\eta = \eta 1 + \eta 2 \tag{8}$$

$$\underline{N} = N1 + N2 \tag{1}$$

Using the example source code from Figure 2, the number of unique operators is $\eta 1 = 11$. The number of unique operands is $\eta 2 = 5$. N1 was 23, and N2 was 18. From these values, the estimated value of Effort is <u>E</u> = 3247.

The $\underline{E}$ metric was originally used to relate the actual time a programmer would take to implement a program. This was questioned by other researchers when they realized that this suggests an arbitrary limit on the mental capacity of all programmers (Shen and others, 1983:156). While there is little evidence supporting the claim that $\underline{E}$ can predict the time to implement a program, there is empirical evidence that this metric correctly estimates maintenance effort and the number of errors in modules (Shen and others, 1983:162 and Henry and others, 1983:130). Discussing studies conducted at Purdue and at General Electric, Shen claims "these two studies tentatively support the conclusion that a program with a lower E measure is easier to comprehend that an equivalent program with a higher E value" (Shen and others, 1983:162).

Hansen's Pair (Cyclomatic Number, Operator Count). Shortly after Myers suggested his extension to the cyclomatic complexity metric, Hansen came up with a different way to modify cyclomatic complexity to get better data. He wrote that while Myers was correct about the differences in complexity between multiple conditions in the same branch statement and a branch statement with only one condition, he stated that the difference was not relevant because no matter how many conditions the branch has, it is going to one location or the other (Hansen, 1978:30).

Hansen decided to not extend cyclomatic complexity, but to use a size metric also. After experimenting, he decided that the unique operator count was the best in combination with cyclomatic complexity (Hansen, 1978:33). He did not include any empirical evidence that he had validated the method, though.

Oviedo's model of program complexity. Oviedo developed a composite metric that measures both control flow complexity and data flow complexity, and reports total program complexity as the sum of the two. Control flow "cf" complexity is calculated as the number of edges in a control flow graph (Oviedo,

1980:148). Data flow "df" complexity will be explained in a following paragraph. The program complexity (C) is calculated as (Harrison and others, 1982:76):

$$C = \alpha cf + \beta df \tag{9}$$

where α and β are weighting factors, which are set to one (Oviedo, 1980:151).

To understand "df", two terms must be defined. A variable is "locally available" for a block if the variable has been defined within the block. A variable is "locally exposed" if it is referenced in a block but it has not been defined yet in the block. The "df" of a node or block N_i is defined as "the number of prior definitions of locally exposed variables in N_i that can reach N_i" (Harrison and others, 1982:76). Figure 8 from Harrison shows code that will be used for an example "df" calculation, along with its program flow graph (Harrison and others, 1982:76).

The "df" of N_0 is always 0, because no prior definitions can reach this block. The two nodes N_1 and N_2 each have "df" of 0, because they are assignment statements that use constants, and no variables are locally exposed. The node N_3 has three locally exposed variables, x, j, and k. Each of these exposed variables has been defined twice before node N_3, so node N_3 has a "df" of 6. Adding the cumulative "df" of all nodes gives an overall "df" of six, as $df_0 = df_1 = df_2 = 0$. As the control flow graph has four edges, "cf" is four. The overall program complexity "C" = 10.

The Oviedo program complexity has the same limitations as other control flow graph metrics. The size complexity of any node will not be measured, and establishing a weighting factor for α will be difficult (Harrison and others, 1982:78). No empirical evidence has been reported to show how well this combination of control structure and data structure metrics work together.

```
                 READ n, x, k
Node 0           If n = 1 then
                    x := 1
Node 1              j := 2
                    m := 5
                 ELSE
                    k := 1
Node 2              j := 3
                 ENDIF
Node 3           d := x + j + k
```

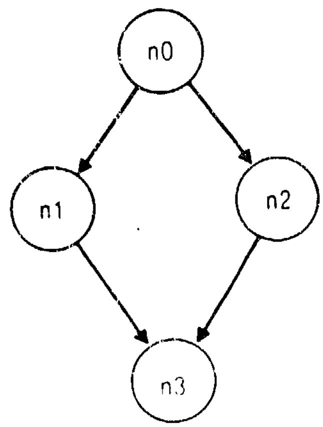

Figure 8. Program Complexity Example
(Harrison and others, 1982:76)

Summary

This chapter has presented examples of size, structure, and composite metrics. The metrics shown are representative of the three different types of metric.

The size metrics are easy to calculate from source code, but measure complexity by considering that what constructs the programs are developed from are irrelevant, only the number of these constructs is important. These metrics

cannot be used until far into the software development cycle, as they need actual source code. This means that they are not good as design tools, but they can show which procedures will have the greatest number of changes and errors during software test and maintenance.

Some data structure metrics can be used earlier in the design cycle, which can give early feedback to the quality of the software. The structure metrics are better able to test the structure of algorithms and data structures, which are the basic framework of all programs, as Niklaus Wirth suggests by the title of his classic book Algorithms + Data Structures = Programs.

Composite metrics are apparently not in common use. But a careful selection of different types of metrics that can complement each others' weaknesses can give a software engineer an insight into the program structure and possible problem areas that no single metric can.

III. Metric Selection Criteria

Introduction

This chapter describes a set of guiding properties that were used to evaluate software complexity metrics. These properties are presented as guidelines to determine how well individual metrics measure complexity, and therefore measure maintainability. Two benefits are derived from comparing the metrics to these criteria: the metrics that more completely measure complexity are identified, and the characteristics each metric best reflects are indicated.

The utility of the first benefit is obvious, but the need for the second requires explanation. If no single metric meets all criteria, metrics that complement each other can be used instead. Using these criteria to screen the metrics shows how each metric can best be applied. As Kearney says, "the properties of a metric determine the ways in which it can be used" (Kearney and others, 1986:1046). If, for example, a single metric that measures control flow complexity does not measure data flow complexity, it can be combined with another metric that meets the data flow complexity criteria and more complete coverage will result.

After the presentation of these criteria, a comparison shows which metrics meet each criterion. The metrics that best fit all of the criteria are discussed. Following that section, a summary of the selection guidelines and the metrics is presented.

Criteria for the Selection of a Maintainability Metric

These criteria are basic guidelines to determine how well a metric measures complexity. These guidelines are loosely arranged into three overall groups. The first five criteria are generic and could be used to evaluate other

types of metrics, such as productivity metrics. They do not determine how well a metric reflects complexity; they verify that the metric is generally applicable across different software. Three criteria are presented to rate how well metrics measure a program's control flow complexity. The last four criteria indicate a metric's measurement of a module's data flow complexity.

These guidelines are not equally important in the measuring of a complexity metric. For example, the two criteria "Ranking Basic Control Structures" and "Nesting and Compound Conditions" are both contained in the criterion "Accurately Reflect Control Flow". They are explicitly enumerated because each criterion is important, but neither is as weighty a consideration as overall control flow.

Although the criteria are not equally important, a definitive weighting of the criteria's relative importance is not given, except for the implied subordination within the control flow and data flow complexity sections. Research to date does not suggest any obvious ranking of criteria. Therefore, the criteria are arbitrarily being considered equally. With this constraint, any metric that satisfies more criteria than another will be judged to better measure complexity.

Clear and Unambiguous. This criteria determines how easily the metric can be evaluated and how easily the result of the evaluation can be compared to results of other evaluations. As Conte expresses, "does the metric lead to a simple result that is easily interpreted" (Conte and others, 1986:22)? The metric should be clear and unambiguous so it can be calculated from just the source code (Levitin, 1986:314).

Lines of code (LOC) is an example of a metric that is not clear and unambiguous. While it may appear to be very easy to count, many researchers

have different counting strategies. This can lead to different LOC values from the same source code.

Halstead's N metric is another metric that appears to be simple to calculate, but presents some difficulty in its calculation. Determination of which tokens should be counted as operators and which tokens should be counted as operands is not always clear. Even after a counting strategy has been defined and adhered to, some function calls defy analysis because they act as both operators and operands.

Intuitive. A metric should be intuitively appealing. It should correspond to a user's innate perception of a program's complexity. The complexity value determined for a less complex module should be less than that of an obviously more complex module. It must always be positive and additive (Levitin, 1986:314; Jayaprakash and others, 1987:241).

For example, if two distinct pieces of code are combined, the complexity value for the joined code should be greater than the complexity value for either piece. An optimum solution occurs when the complexity value for the joined code equals the sum of the complexity values for the separate pieces.

Language Independent. A metric that is based on a single language is not generally applicable. A metric should be as universally applicable as possible so that it can be used to evaluate software written in any programming language (Jayaprakash and others, 1987:241).

A metric that estimates complexity based on the number of GOTO statements in a program may be a valuable quality measurement tool when used with the FORTRAN language. This metric would be worthless when used with the Prolog language, as Prolog does not have any GOTO statements. This metric would also be of limited utility in measuring the complexity of modules written in Pascal or Ada. While both of these languages allow a GOTO statement, such

use is heavily discouraged. Therefore, the likelihood of determining a reasonable value for a Pascal module's complexity using this metric is negligible.

The FORTRAN language allows only a single statement to be placed on a line. This simplifies the calculation of LOC. Other languages such as JOVIAL, Pascal, and C have special characters that delimit the end of a statement. With these languages, several statements can be placed on one line. This can complicate calculation of LOC for modules written in these languages. For this reason, LOC is not a language independent metric.

Prescriptive. A complexity metric should not only measure the software's complexity, it should also reveal how the software should be modified to minimize complexity (Kearney and others, 1986:1047). The metric's results should direct the software's maintainers to the modules that need to be changed, and it should reveal to the maintainers what changes need to be made.

If a module has a large value for information flow (INFO), that should suggest to a developer or maintainer that the module needs to be further decomposed into more manageable modules, and the inter-module communications should be simplified. A possible complication is if a software developer wants to attain the smallest value for INFO, he can write a program as a single module with no interconnections between modules. This programming practice would increase the complexity of the program, not decrease it as the INFO value would lead us to believe.

Robustness. A trivial reordering of the program's statements should not lessen the complexity the metric reflects. A reduction in a metric's measurement should result from an improvement in the program measured (Kearney and others, 1986:1047). Adherence to this criteria forces any programming practice that reduces the metric value to also reduce the program's

complexity. As Conte asked, "is the metric sensitive to the artificial manipulation of some factors that do not affect the performance of the software" (Conte and others, 1986:22)?

As an example, a program that scored high on McCabe's cyclomatic complexity measure because of the number of loops could be rewritten with the loops as in-line code. This would lessen the cyclomatic complexity score, but might significantly increase the complexity of the module.

Accurately Reflect Control Flow. The control flow in a program is the path through a program that is followed during execution. By measuring the number of paths through a module, a determination can be made if the module is becoming difficult to understand and should be partitioned into separate, smaller modules (McCabe, 1983:3). A satisfactory control flow metric should measure how easily understandable the control structures are in a program.

Ranking Basic Control Structures. Structured programming methodology recognizes three basic control constructs: sequential, selection, and repetition constructs (Prather, 1984:341). The sequential statements have the lowest control complexity, as the flow of control is always to the immediately following statement. Selection statements can branch to one or more other statements. In a selection statement's basic form, either one section of code or another is executed. Whichever section completes execution, control flow continues from the same point. A repetition statement also has two locations from which it can continue execution. Either control goes to the statement following the bottom of the loop, or control passes back to the top of the loop and the statements within the loop are executed repeatedly.

Jayaprakash recommends that any control flow metric should show that sequential statements are less complex than single-selection selection statements, which are less complex than repetition statements (Jayaprakash and

others, 1987:241). Repetition statements should be counted as more complex than selection statements because they cause backwards branches in the code, and "it is well known that these [backwards branches] cause the most difficulty in practice" (Prather, 1984:345).

Nesting and Compound Conditions. A complexity measure should be sensitive to nesting in branch statements. Several researchers have stated that one module that contains two selection statements with one nested inside the other is more complex than a different module that has the identical two conditions occurring in sequence (Jayaprakash and others, 1987:242; Myers, 1977:62). A selection statement that has a compound condition is slightly more complex than a selection statement with only a single condition; this added complexity should be reflected by a complexity metric.

Accurately Reflect Data Flow. Another factor that impacts complexity is the data flow into, within, and out of the module. To better understand the module's complexity, the complexity of this data flow should be measured in addition to the module's control flow complexity. According to Harrison, "another factor that influences software complexity is the configuration and use of data within the program. Several methods can be used to measure complexity by the way program data are used, organized, or allocated" (Harrison and others, 1982:67).

Indicates Data Amount. A basic factor that determines the complexity of the data flows within a module is the amount of data that a maintainer has to comprehend. A large number of variables that must be understood can make the maintainer's assignment very difficult. These include the number of variable parameters and global data flowing into and out of a module, and the variables declared and used locally to the module.

Shows Data Use. How the variables are actually used in a module is another important determiner of module complexity. Determining which variable is modified and where it is modified can be arduous in a long module. If a variable is used and modified within a small portion of a module, the variable will be less challenging to remember. If, conversely, a variable is set once at the beginning of a module and not used for 100 lines, the maintainer may have a problem remembering the variable's value.

Reflects Inter-Module Data Links. The coupling of the module is reflected by the number of data links into and out of the module. Measuring the data links is important because "by observing the patterns of communication among the system components we are in a position to define measurements for complexity, module coupling, level interactions, and stress points" (Henry and Kafura, 1981:511).

Comparison of Metrics by Selection Criteria.

Figure 9 shows relationships between metrics described in Chapter Two and the selection criteria developed previously. This is presented in a table to better show the relationships between size metrics and general complexity, data structure metrics and data complexity, control structure metrics and control complexity, and the hybrid metrics.

The metric selection criteria are shown across the top of the grid, in the same order as they were described previously in this chapter. They are separated into three groups just as their descriptions were. The metrics are presented in the order they were described in Chapter Two. They are shown in the same groups that were established in Chapter Two: size metrics, data structure metrics, control structure metrics, and then hybrid metrics.

To show that a metric satisfies a criterion, a mark is placed in the box at the intersection of the criterion's column and the metric's row. No mark implies that the metric either does not satisfy the criterion at all, or it does so poorly. An indication of partial agreement (*) reflects that the metric does not fully meet the criterion, but it does help measure it. An indication of substantial agreement (!) suggests that the metric fully satisfies the criterion. Justification for the indications is given in Appendix A, Justification for Metric Complexity Criteria Ratings.

Figure 9 shows that size measures reflect neither control structure complexity nor data structure complexity. Neither data structure metric can measure control structure complexity, nor can any control structure metric measure data structure complexity. Oviedo's "C" hybrid metric, which measures both control structure complexity and data structure complexity, measures the best across the whole spectrum. If a single metric from the above list had to be chosen, this metric fits the criteria better than any other.

The most complete control flow complexity measure is MEBOW, which has more agreement indications than "C". The most complete data flow complexity measure is INFO, which has more substantially-agree indications than "C" does within the data flow criteria. A combination of MEBOW and INFO has substantially-agree marks in eight categories and partial-agree marks in three of the other four criteria. The only criteria that neither MEBOW nor INFO meet is the "Shows Data Use" criteria. Because of the large coverage of metric selection criteria, a hybrid metric using both MEBOW and INFO in a 2-dimensional vector is suggested.

A strong case can be made for using a combination of the "C" metric and INFO. The control flow complexity would be measured nearly as well, and the data flow complexity would be measured by all criteria. But the "C" metric

	Clear	Intuitive	Language	Prescriptive	Robust	Control	Rank	Nest	Data	Amount	Use	Link
Size Metrics												
LOC		!		*	*							
N		!		*	*							
Data Metrics												
Span	!	!	!	*	*						*	
INFO	!	!	!	*	*				*	!		!
Control Metrics												
v(G)	*	*	!	*		*	*					
Knot	*	*	*	*	!	*		*				
MEBOW	*	*	!	!	!	*	!	*				
Hybrid Metrics												
E		*		*	*							
v(G), $\eta 1$	!	*	*	!	*	*	*					
C		*	*	*	!	*	*	!	*	*	!	

(!) indicates substantial agreement (*) indicates partial agreement

Figure 9. Metrics vs. Metric Selection Criteria

lacks empirical evidence to show it actually measures complexity. MEBOW itself has not been tested, but it has a substantial weight of evidence supporting the use of v(G), which is a component of MEBOW. Knot count also has empirical evidence that it measures program complexity (Woodward and others, 1983:105), although no studies have shown a correlation between knot count and program mainainability. Because of the amount of evidence supporting MEBOW's components, it is being recommended instead of "C".

Summary

The purpose of this chapter was to define a set of criteria that would help determine which metrics are more useful than others. These criteria were grouped into three categories, the general applicability criteria, the control flow complexity criteria, and the data flow complexity criteria. Each of these criteria was explained, and justifications why each is important were given.

This list of metric selection guidelines is not a complete set of possible properties that a metric might have. As Kearney stated, "although the preceding list of properties may be flawed, it is essential that the designers and users of software complexity measures recognize that the properties of measures constrain their usefulness and applicability" (Kearney and others, 1986:1048). Overall, it must be remembered that the selection of a metric to measure maintainability and complexity was the desired end result. The guidelines were chosen with that in mind.

Once a set of criteria for determining how well a metric measures complexity was defined, they were used to gauge the metrics. By comparing each metric with each criteria, the metrics that had the most complete coverage in each criteria group were uncovered. A simple comparison of the maximum number of criteria successfully matched by the metrics brought the candidate

metrics down to two pairs. "C" was disqualified because of its lack of empirical support.

IV. Maintainability Metrics Proposed for AFOTEC Use

Introduction

In the previous chapter, two metrics were selected for use in measuring maintainability. These metrics measure different aspects of software complexity, so a combination of these two metrics will be more comprehensive than either. This chapter explains these metrics in greater detail than they were discussed in Chapter Two.

This explanation includes data that researchers have obtained during various studies. Further empirical data that supports the use of a hybrid metric for the measurement of complexity by comparing the metric value to a module's error count is given in Appendix C, Empirical Support for Hybrid Metrics. Further discussion of the theoretical support for MEBOW and information flow follows. The data both support and repudiate the use of MEBOW, in the form of cyclomatic complexity, and information flow. Both sides of the issue are presented, and justification why the evidence supports the metrics' use more than the evidence against the metrics precludes their use is given.

After considering the empirical support for hybrid metrics in general, and MEBOW with information flow in particular, metric implementation considerations are presented. The problems associated with parsing source code for various languages will be considered. Then rules for calculating the metrics will be given. An example of this calculation is given in Appendix D, Calculation of Metric Value for an Ada Procedure.

The next discussion centers around the determination of a threshold value, and the validation of the metrics. A method using test cases to

determine threshold ranges is given. A plan to determine how well the metrics measure maintainability is presented for AFOTEC's use.

Proposed Maintainability Metrics

As the last chapter explained, the two metrics MEBOW and information flow (INFO) were selected. MEBOW met as many control flow and general complexity criteria as any other metric presented. No research has been located that provides empirical evidence that MEBOW measures either complexity or maintainability. This metric was proposed in 1987 and no known studies have been completed to determine the worth of this metric. Fortunately, MEBOW's component metrics, cyclomatic complexity (v(G)) and knot count (KNOT), each have empirical support as complexity measures. The authors of MEBOW analytically argue why the combination of v(G) and KNOT is a better measure of complexity than either metric is by itself.

INFO met as many data flow and general complexity criteria as any other metric presented. INFO also has empirical support as a measure of complexity. Henry and Kafura also argue that INFO would describe complexity better if it were combined with another complexity metric, such as Halstead's length or v(G) (Henry and Kafura, 1981:513).

The main point of this thesis is that both of these metrics should be calculated for each module of a program. These resulting scores should be compared to determine which modules are more complex than others, and the scores should be compared to a threshold to judge which modules are difficult to maintain. This evaluation of each module separately, instead of the program as a whole, follows the guidelines given in the Vol. 3. That this module-by-module examination is useful and has ample support. Henry and Kafura state that "the complexity of a module is defined to be the sum of the complexities of the procedures within the module. It is interesting to note that the majority of

a module's complexity is due to a few very complex procedures" (Henry and Kafura, 1981:514). Basili and Perricone have evidence that errors are usually confined to a single module, so maintenance efforts will only have to modify a single module. They assert:

> It was found that 89 percent of the errors could be corrected by changing only one module. This is a good argument for the modularity of the software. It also shows that there is not a large amount of interdependence among the modules with respect to an error [Basili and Perricone, 1984:45].

Justification of Metrics Selected

This section will explain why the MEBOW and INFO metrics should be combined into a hybrid metric. First, a discussion of why a hybrid metric should be used is presented. This discussion is followed by evidence that shows how hybrid metrics were better able to measure complexity and maintainability than other metrics.

The following section expresses arguments given to explain why MEBOW will be better than either v(G) or KNOT. This section presents empirical evidence that the two metrics measure complexity. Research that supports the use of INFO is examined in the next section.

Hybrid Metric Benefits and Detriments. Chapter Two included a discussion about what a hybrid or composite metric is. Some evidence was presented that supported the use of a hybrid metric. The suggestion made that the use of metrics from different metric classes together can provide a greater insight into a module's complexity was a main topic of the chapter.

This section expands upon the previous discussion of hybrid metrics. Four studies are presented that conclude that hybrid metrics can better measure complexity than metrics from a single class. These studies were accomplished by Kafura and Canning (1985), Harrison and Cook (1987), Ramamurthy and Melton

(1986), and Li and Cheung (1987). The metrics tested and the conclusions drawn from the studies are examined in this section. In the next section, data from the four studies is presented.

Kafura and Canning prepared a study of three production software systems written in FORTRAN. These software systems were from NASA and an extensive database of development information was kept for each system for use at the Software Engineering Laboratory. The Software Engineering Laboratory is an organization composed of three members: NASA/Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation. This association monitors the details of software development for later examination. The information that Kafura and Canning used were the counts of component errors and component coding time for each module (Kafura and Canning, 1985:380). These data were used in an attempt to validate the use of ten metrics.

The ten metrics that were used to analyze the software systems were placed into three groups: code metrics, structure metrics, and hybrid metrics. The three metrics they considered to be code metrics were LOC, Halstead's effort (E), and v(G). Three of the four metrics in the structure metrics category were not explained in Chapter Two, and as they did not greatly affect the study, they will not be considered here. The fourth structure metric used was INFO. The hybrid metrics were a combination of LOC and three of the structure metrics, including INFO-LOC.

They first obtained results to see if "significant differences in software metric values are related to corresponding differences in errors and/or effort" (Ibid:380). These results showed some correlation between the metric values and the errors and coding times. Then the combined coding time and error factors were compared to the metrics, and better correlations resulted. This prompted

the researchers to assert "the observations made above lead us to conclude that growth in the metric values corresponds to increases individually in error proneness and coding time requirements and that this trend becomes more sharply defined when the combination of error and coding time is taken into account. This is both a validation of the metrics and a motivation to use multiple resource [error and coding time data] variables in further validations" (ibid:381).

They accomplished the measurement of combined errors and coding time by separating their components (modules) into categories of higher and lower errors and coding time. Those components that were high in both error count and coding time were termed "difficult," and those with low error counts and coding time were termed "easy" components. With these categories, they explained their higher correlations with "it is important to consider the combination of these factors because ... a component with a high metric value may result in few errors because a large amount of time was invested in the coding of this component" (ibid:381).

These "difficult" components were then separated into categories by order of difficulty. These categories were those components that were one or more standard deviations above the mean for number of errors and coding time. An outlier is a component more than one standard deviation above the mean, and an extreme outlier is a component more than two above. The ten metrics were used to determine how many outlier error components they could identify. Their results indicate that the metric that best identified the outliers and extreme outliers was the hybrid metric INFO-LOC. This led to their conclusion that "this observation is significant because it supports the need to use metrics from all classes, confirms again that structure and code metrics are measuring different properties of software components" (ibid:382). They also suggested the

use of a "minimum metric count of 4", which increases the number of outlier error components that can be identified (ibid:384).

This research supports the use of a hybrid metric. Some issues other than the use of a hybrid metric were introduced. Their outliers were for total error and total coding time. Coding time has not been presented within this thesis as a factor within maintainability, but a careful consideration of maintainability would support its use.

Harrison and Cook were developing a metric that would measure an entire software system's complexity. They considered metrics in two categories, macrolevel measures and microlevel measures. The macrolevel measures consider how the function of each module fits into the overall system. The microlevel measures consider the detailed operation of a single module. They described a new metric that would embrace both macrolevel and microlevel complexity (Harrison and Cook, 1987:213).

They called their new metric MMC, for Macro/Micro Complexity. This metric includes both a macrolevel and a microlevel component. The macrolevel measure uses the number of global and parameter variables that are used within a module, much like INFO does. It also includes a calculation of "quality of the subprogram's documentation" (ibid:216), which is simply a ratio of the number of comments to the number of source lines within each module. The microlevel measure they used was v(G). The sum of the microcomplexity of each module and the amount it contributes to the overall complexity through its use of data is the MMC.

This hybrid measure was compared to six other metrics, using error data from a 30,000-line compiler project written in C. MMC had a higher correlation with the number of errors that occurred within each module than any other

metric (ibid:217). One of the other metrics was v(G). Their results show that adding a data flow component to v(G) will create a metric that can better detect modules that will have the greatest number of errors. This also supports the use of a hybrid metric to measure software complexity.

Ramamurthy and Melton looked at Hansen's ordered pair of v(G) and operator count and decided that a combination of Halstead's and McCabe's metrics would make a good metric. They combined the two metrics into a single metric to prevent the problem that occurs when the two metrics give conflicting reports about the relative complexities of two modules. They weighted the count of certain operators and operands by the level of nesting. They defined a value "C", which is one greater than the current structure's level of nesting. They "call C the cyclomatic complexity of the control structure" (Ramamurthy and Melton, 1986:310). In addition to counting all of the operands and operators, they counted those that were part of a control structure and added the value of the nesting level to the count of that operator or operand. This gave certain operators greater weight than others.

To test these weighted metrics, they calculated the value of the unweighted and the weighted metrics for a number of test programs. These test programs were in three groups: programs with the same Software Science values but different v(G), programs with the same v(G) but different Software Science values, and a general collection of programs. Their results showed that "the weighted metrics do detect the complexities which the software science metrics detect and the complexities which the cyclomatic number detects" (ibid:313). These results also support the use of a hybrid metric from two different classes of metric.

Li and Cheung compared 31 metrics to see if generalizations could be made about different classes of metrics. These metrics were calculated for 255

student assignments in FORTRAN. No attempt was made to compare any of the metrics to any "external validation," such as a comparison of the metrics to the numbers of errors or the amount of time required to code each assignment (Li and Cheung, 1987:707). This study was just to consider the general relationships among the different metrics and determine the internal consistency among different measures within the same metric (Software Science).

They considered the correlations among many different metrics, within the same classes and between classes. One of their conclusions was that a hybrid metric would be very useful. They said:

> In general, the control flow metrics fail to be comprehensive and do not consider the contribution of any factor except control flow complexity. However, these metrics can differentiate between two programs of similar VOLUME metrics and certainly are related to the software quality. Hence, a useful approach is to use VOLUME metrics for prior classification and then to use CONTROL ORGANIZATION measures to evaluate the programs in detail [Li and Cheung, 1987:707].

In a different study, Kafura and Reddy came to the same conclusion. They stated their study "has also confirmed the results obtained in previous work with respect to the distinction between the code and structure metrics. This distinction was evident in that the maintenance changes to components might dramatically alter the values of metrics in one class of metrics without changing materially the values of metrics in the other class" (Kafura and Reddy, 1987:342).

The results of these four studies suggest that the use of a hybrid metric is a useful technique for measuring complexity. Appendix C, Empirical Support for Hybrid Metrics, presents the data that three of these studies generated to support their conclusions.

MEBOW. MEBOW was designed as a comprehensive control flow complexity metric that did not have the deficiencies of other popular metrics. No empirical

evidence or examples were shown to support that this proposed metric measured complexity better than the measures it was supposed to supplant. Instead, this metric was shown to meet twelve "precisely-stated intuitive properties expected of any control flow complexity metric" (Jayaprakash and others, 1987:238). These properties included such factors as language independency, ranking of basic control constructs, and sensitivity to nesting which were presented in the previous chapter as metric selection criteria. None of the other control flow complexity metrics presented (v(G), KNOT, and SCOPE ratio) was able to satisfy all twelve properties. Jayaprakash, Lakshmanan, and Sinha explained that these properties were important by stating "the idea is that if a control flow complexity metric fails to satisfy these intuitive properties, any extent of empirical evidence supporting its use in estimating the maintenance cost of software, or predicting the number of errors in the program, etc., cannot really provide enough confidence for its widespread use in practice" (ibid:238).

They consider v(G) to be only a special case of MEBOW, where each branch is counted as one, and a "constant bias" of 2 (for 2p) is added to calculate the value (ibid:240). KNOT is also considered as a special case of MEBOW, where knots are given a weight of one, and all other control flow factors are ignored. Because these two metrics are encompassed by MEBOW, they believe that MEBOW will better measure complexity than either metric could alone. They state, "it appears, therefore, that by suitably assigning the relative weights to the factors stated above, it is possible to arrive at a complexity metric which combines the strengths of the existing measures" (ibid:240).

After using Jayaprakash, Lakshmanan, and Sinha's arguments that MEBOW is better than either v(G) or KNOT, some evidence of how well these two metrics

measure complexity is given for comparison. Many studies have attempted to measure how well v(G) measures complexity. Some of the conclusions and results of these studies are presented. KNOT has not been studied in as much detail, but some results for KNOT are shown.

Evidence Supporting the Use of v(G). In a pair of experiments, Curtis attempted to measure the psychological complexity of software maintenance tasks by comparing Halstead's effort (E) and v(G) to the actual performance of programmers on two software maintenance tasks. The programmer's performance was measured in two ways. The first was based on the premise that a good measure of a programmer's understanding of a program is his ability to learn its function and reproduce an equivalent program without notes. This performance was measured by the "functional correctness of each separately reconstructed statement" (Curtis, and others, 1980a:297). The second performance criterion was measured by how correctly a requested change was implemented and the time to perform the modification. Measurements of the accuracy of implementation and time to completion were correlated between modules and their E and v(G) values. The correlations were not high, but the study's conclusions were "the two experiments comprising this study produced empirical evidence that software complexity metrics were related to difficulty programmers experienced in understanding and modifying software" (ibid:301). These conclusions were questionable, as the correlations for v(G) ranged from -.55 to -.21 in the first experiment, and from .38 to -.36 in the second experiment.

In a later experiment, Curtis found v(G) to be a better predictor of programmer performance. This experiment measured how long each of 54 professional programmers took to find and correct a single error in three separate FORTRAN programs. The correlations between v(G) and the average

performance were given to be .63 for single subroutines, and .65 for total programs. These correlations were much stronger than in the previous experiments. These results "demonstrated that far stronger results could be obtained when the limitations in our earlier experimental procedures were overcome. For instance, our previous research was conducted exclusively on small-sized (35--55 lines of code) programs, which seems to have limited those results..." (Curtis and others, 1980b:307).

McCabe stated that a v(G) of ten is a reasonable upper limit for a single procedure, and if complexity exceeds ten, the procedure should be decomposed into smaller procedures. Walsh studied software developed for the AEGIS Naval Weapon System radar to determine if procedures with higher v(G) had a higher number of errors. He quickly determined a correlation between those procedures with a high v(G) and the occurrence of errors in those procedures. But he saw that those were also the largest procedures in terms of lines of code. To determine if the number of decisions within a procedure had an impact, and that v(G) was not just measuring size, Walsh separated those modules with v(G) of ten or more and those with a lower v(G) to compare their relative error count. He found that the procedures with v(G) of ten or more had 21 percent more errors per 100 lines of code than those with a smaller v(G). Those procedures with higher v(G) averaged 5.60 errors per 100 source statements, while the others averaged 4.59 errors (Walsh, 1983:95). This suggests that v(G) is measuring something more than just the size of a procedure, and that it is valuable in predicting the error rate for software. Walsh explained his numbers by stating:

> As the number of detected errors in a piece of software increases, the probability of the existence of more undetected errors also increases. Put simply, errors come in clusters. Thus, it can be confidently predicted that when the procedures in the study enter

the maintenance phase of their existence, the procedures with a complexity greater than or equal to ten will continue to experience higher error rates than those procedures with complexity below ten [Walsh, 1983:95-96].

Harrison and Cook's macrocomplexity and microcomplexity metrics were described earlier. Their correlations for v(G) with error occurrence and other metrics are presented in Figure 15 within Appendix C. They also compared how well each metric was able to identify the most error prone and least error prone modules. The modules were listed from most errors discovered to least, and a comparison was made to how well the metrics were able to rank order the twenty modules. The correlation for how well v(G)'s ranking of all the modules matched their actual error ranking was only .50, but a ranking of just the most error prone six and least error prone six modules was .81. Both numbers were the middle scores for the seven metrics measured. A conclusion that Harrison and Cook drew from that data is "this suggests that the metrics work quite well in identifying the 'extraordinary cases,' but do a relatively poor job of distinguishing among modules which do not fit into one of these 'extraordinary' categories (i.e., either few errors or many)" (Harrison and Cook, 1987:218). If a software manager had this type of error prediction data from v(G), he could spend more resources testing the more complex modules, reducing resources on those modules that are determined to be less complex.

Shepperd studied v(G) and the results of other research and came to a different conclusion about the metric than other researchers. He felt the metric is based on "poor theoretical grounds and an inadequate model of software development" (Shepperd, 1988:30). He disputes the metric's empirical validation results, also. He disagrees that an intuitive appeal should be part of a metric's validation and sneeringly discards intuition as a factor in the consideration of the metric. While it may be understood that a metric's intuitive appeal should

not be its only justification for use, this factor should not be offhandedly dismissed.

A list of theoretical objections was presented by Shepperd. One is that "the treatment of case statements has also been subject to disagreement" (ibid:32). He discusses that different researchers have used different counting strategies to number the decisions in a case statement. This is one reason that a counting strategy should be rigorously defined and adhered to. Shepperd points out that v(G) cannot measure the complexity of sequential statements. This is certainly a valid criticism and is why others have added a size complexity metric to v(G).

According to some researchers mentioned by Shepperd, "applying generally accepted techniques to improve program structure" can actually increase v(G) (ibid:32). This is because the metric is insensitive to the unstructuredness of a program, as it only counts decisions and does not reflect if a decision causes jumping out of loops or into another decision, which are generally considered to be unstructured techniques. This is a reason why KNOT should be added to v(G), so that less structured techniques will cause higher complexity values. This is one of the justifications for MEBOW. As was mentioned in Chapter Two, Shepperd believes that v(G) does not measure inter-module complexity well. Since this appears to be a correct appraisal, this criticism is one reason why MEBOW calculation is being suggested for intra-module use only.

Shepperd refers to some data that Evangelist reported showing "the application of only 2 out of 26 of Kernighan and Plauger's rules of good programming style invariably results in a decrease in cyclomatic complexity" (ibid:32). This contradicts Myers' results comparing the v(G) of more and less structured code from Kernighan and Plauger's The Elements of Programming Style. According to Myers' calculations, v(G) was always lower for what was

subjectively considered the more structured code from their examples (Myers, 1977:64).

Shepperd reasoned that even though he did not agree with the theoretical justifications for v(G), it would still be a useful metric if it can be shown to accurately measure complexity. "The theoretical objections to the metric, that it ignores other aspects of software such as data and functional complexity, are not necessarily fatal. It is easy to construct certain pathological examples, but this need not invalidate the metric if it is possible to demonstrate that in practice it provides a useful engineering predictor of factors that are associated with complexity" (ibid:33). He then proceeded to condemn other researchers' experimental methods, statistical correlation techniques, and results, ending with a sweeping statement that "to summarize, many of the empirical validations of McCabe's metric need to be interpreted with caution" (ibid:34). His points were well made, but he did not refute all of the results he presented that suggest that v(G) could determine error-prone modules. His contention that a problem with this metric is that it does not measure data flow complexity is a further argument that v(G) should be combined with a data flow complexity metric such as INFO.

Evidence Supporting the Use of KNOT. Woodward, Hennel, and Hedley did not show any data that compared KNOT to any error data or the time it took programmers to modify a module. Instead, they showed examples of two code fragments that performed the same function and stated the source code with the smaller KNOT count was more structured. These same examples showed that v(G) did not change. While the source code with fewer knots was typically shorter and had fewer branches than the example with more knots, one example

they showed had a lower KNOT count for unstructured code, while the structured version had a higher KNOT count (Howatt, 1988).

One difference between KNOT count and v(G) is that the KNOT count does not measure the structure of the program's control flow. Instead, it measures the structure of the source code. This can be considered a benefit because a program's maintainer will work with the source code and not a flow graph (Howatt, 1988).

Considering that a more structured version of a program is better than a less structured version, adding KNOT to v(G) seems to be a better way to measure a program's structuredness than v(G) alone, in addition to determining how difficult it will be to test. Figure 10 shows a more structured version of the code fragment from Figure 7 (in Chapter Two). These code fragments are identical in function, but this fragment is more structured than the other. The v(G) of both is three, but the second version has only three knots and has no backwards jumps. Discussing the benefits of KNOT, they state "we feel that the knot count provides a much clearer indication of program readability... The high knot counts for these [two other] routines confirm not only the visual impression of high complexity but also the difficulty actually encountered in translating them to other languages" (Woodward and others, 1983:105).

In Li and Cheung's comparison with 17 other metrics, they assert v(G) "correlates well with Halstead's, Gilb's, Knot Counts, SCOPE, EDGES, and NODES metrics. So, the cyclomatic complexity metric seems to bridge the gap between the two categories: VOLUME and CONTROL ORGANIZATION metrics" (Li and Cheung, 1987:705). Their data shows v(G) correlations in the range of .971 to .796 with the 17 other metrics (ibid:704). Knot count has correlations in the range of .948 to .799 with the same metrics. These correlations suggest that these two metrics measure complexity as well as any other established metrics.

```
      CALL TPR
      IF (ZR) 500, 500, 100
>100  CALL TED
      IF (Z3) 200 200 550
>200  ZG = ZG + 1
      ZC = 0
      CALL TCO
      GOTO 600
>500  Z3 = 1
>550  CALL TEC
      ZB = ZB + 1
      ZC = ZC + 1
>600  CALL TRA
      RETURN
      END
```

= KNOT

Figure 10. More Structured Knot Example
(Woodward and others, 1983:104)

In this section, much conflicting data and data interpretations have been presented. While the use of McCabe's cyclomatic complexity metric is now popular, the metric has some obvious limitations. It appears that many of the theoretical objections that Shepperd has against v(G), such as it not being able to measure structuredness or data flow, would be remedied by adding KNOT count with the use of MEBOW, and using INFO to measure data flow. A well-defined counting strategy will lessen the problem of researchers measuring the same modules differently because they are not counting MEBOW in the same fashion. Overall, it appears that v(G) lays a good foundation for the measurement of control complexity, and MEBOW improves upon this foundation. In conclusion, "several variations [for measuring] the cyclomatic complexity metric have shown very encouraging potential for usefulness as measures of software product quality" (Basili and Reiter, 1980:287).

Information Flow. Information flow was described in Chapter Two, and an example was given for INFO calculation. This section presents further evidence that INFO reliably exhibits complexity, in the form of data flow complexity. Basili and Perricone explain why this data complexity is an important factor to measure when they state "interfaces appear to be the major problem, regardless of the module type" when referring to where errors occur most often in a program (Basili and Perricone, 1984:47).

One of the earliest experiments with INFO was done by Henry, Kafura, and Harris (Henry and others, 1983:125). Values for INFO, v(G), and Halstead's *E* were calculated for source code modules from the Unix operating system and were then compared to a list of errors found during the system's development. The three metrics were also compared against each other to see if they appeared to measure the same factors. Their results are summarized in Figure 11. The formula used to calculate INFO is shown as (5) in Chapter Two.

	E	v(C)	INFO
Errors	.89	.96	.95
E		.8411	.3830
v(G)			.3459

Figure 11. A Comparison of Three Metrics (Henry and others, 1983:130)

These results suggest two conclusions. The first is that INFO is a useful measure of complexity, as a high correlation was found with detected errors. The second is that INFO measures different factors than Halstead's *E* and v(G),

as the correlations between INFO and the other two metrics were small. Henry explained this result as "the information flow complexity measurement is orthogonal to the other two metrics since it has a low correlation to both Halstead's and McCabe's metrics. The independence of the information flow metric is explained by its greater concentration on the manner on which system components are interconnected" (ibid:130).

Harrison and Cook's comparison of INFO (they called it HNK) with other metrics and errors appear in Figure 15 in Appendix C. The correlation between INFO and errors is not as high as Henry's results were, with a .62 correlation. An obvious cause of this discrepancy is that they did not have access to all of the information normally used to calculate INFO. Instead, they used the formula:

(fan-in + fan-out) ** 2 * length (11)

Apparently, summing the data factors instead of using their product weakened their influence in the metric calculation. Therefore, a greater correlation with E and v(G) were encountered than before, but a worse correlation with the error count was the result.

INFO, like v(G), was used in an attempt to rank order the most and least error-prone modules used in Harrison and Cook's study. INFO had a .55 correlation with the ranking of all twenty modules used. This is slightly greater than the .50 correlation received by v(G). Measuring just the most error-prone six modules and least error-prone six modules, INFO had a .77 correlation, which is somewhat smaller than the v(G) result of .81 (Harrison and Cook, 1987:218).

Recalling Kafura and Canning's work with INFO, Figure 14 in Appendix C shows that INFO was able to identify extreme outlier error components better

than any non-hybrid metric used. INFO also measured the number of resource outliers better than any other metric except LOC. INFO correctly identified 38/85 error and coding time outliers (Kafura and Canning, 1985:383).

Rodriguez and Tsai used four metrics to determine the complexity of two medium-sized "system implementation packages" (Rodriguez and Tsai, 1986:369). The metrics used were INFO, LOC, v(G), and Halstead's volume metric, which is defined as (Conte and others, 1986:42):

$$\underline{V} = \underline{N} \cdot \log_2 \eta \qquad (12)$$

Just as with the Harrison and Cook experiment, INFO was not calculated as Henry, Kafura, and Harris suggest it should. Rodriguez and Tsai explain, "as a result of our approach, the definition of fan-in and fan-out given by Henry and Kafura has to be revised" (Rodriguez and Tsai, 1986:370). They show that if global variables are modified within a local procedure, they are not counted in the data flow of the overall procedure.

> Using this [Henry and others'] formula, no good correlations of the metrics are found against modifications and errors. However, using Halstead's ideas, an adaptation leads us to formulate the complexity of a procedure as:
>
> length * ln(fan-in * fan-out)
>
> Using this definition of complexity, the correlations found are improved considerably [Ibid:370].

These four metrics were compared to the number of modifications reported for each module throughout the development and maintenance of the two programs. These modifications were adaptive and perfective, rather than corrective in nature. Rather than calculating if each metric can identify the most-modified modules, the study showed how the metrics added together explained the variation in the number of modifications. Their results showed that INFO explained 80.297% of the variation in modifications, while INFO with LOC could explain 84.994% of the variation. With all four metrics combined,

87.257% of the variation of modification could be explained. They concluded "we have to keep in mind that the high regression coefficient (0.87257) shows that meaningful relationships exist between each metric taken individually or jointly and the index of errors of modifications to the software" (ibid:371).

Further analysis was performed to see if they could determine some module size threshold that the four metrics would not correlate well with errors or modifications. Their final conclusions were "all four metrics are useful indicators of the occurrence of errors or future modifications of software units, when the unit size exceeds some threshold. For our study cases, that threshold is 75 lines of code" (ibid:374).

Kitchenham performed a study that compares v(G), LOC, and an INFO-like metric called Information Linkage (IL) with the 226 modules of a communications program. As INFO does, IL considers the number of data flows into and out of a procedure. The number of procedures that call the current procedure are added, as is the number of procedures the current procedure calls. These factors are added, instead of multiplied as INFO does. Kitchenham compares the three metrics with the number of perfective changes and the number of corrective changes made to the communications system. The percentage of error-prone and change-prone procedures that IL identified was lower than the percentages identified by the LOC and v(G) metrics. She suggests that these results directly contradict Kafura and Canning's results, although they calculated INFO much differently than she did (Kitchenham, 1988:374).

Although these results do not support the use of INFO to the extent that other studies do, some interesting conclusions were given:

> The results of this study suggest that it might be a cost-effective procedure to apply more stringent development procedures to programs with high fan-out values. Extra time spent on 12% of the programs would have been 63% efficient (since 63% of the programs

identified warranted additional development time), but would have only been 24% effective (since only 24% of the programs which warranted additional development time would have been identified). Extra time spent on a randomly selected 13% of programs would have been 33% efficient and 13% effective [Kitchenham, 1988:375].

Metric Implementation Considerations

The previous section explained in some detail why hybrid metrics are useful, and presented the results of studies that show how hybrid metrics can measure complexity well. Then MEBOW was described and evidence supporting the use of v(G) and KNOT was given. Finally, study results favoring INFO's use was presented. This section explains some issues of metric implementation such as counting strategies, and determining threshold values.

Calculation of Metric Value. Figure 12 shows a sample FORTRAN program that reads three numbers and writes the greatest of the three numbers. Basic blocks, which are the straight line segments of code, are numbered and separated by lines within the figure. A flowgraph of the program is presented next to the program. Following Jayaprakash's terminology, blocks that have only one source statement are represented in the flowgraph as horizontal lines, and blocks that contain more than one statement are represented as circles. The implicit branches are dotted lines, and explicit branches are solid lines.

According to the definition in Chapter Two, MEBOW is calculated by counting branches and KNOTS and adding their respective weights. The raw weights of the branch types are:

1. Implicit forward branch = 1
2. Implicit backward branch = 3
3. Explicit forward branch = 2
4. Explicit backward branch = 6

These values represent the MEBOW developer's contention that an explicit branch is twice as harmful as an implicit branch, and a backwards branch is three times as harmful as a forward branch. According to the developers, "it is

important to note that the weights associated with each of these entities are only relative to each other and their actual values are of no significance" (ibid:240–241). They also stated that "it also seems meaningful to assign a relatively high weight to each knot since it normally represents an unstructuredness in the program" (ibid:241). Because each KNOT involves two branches, the weight of the KNOT is calculated as the sum of the branches' weights. If two implicit branches intersect, as in an IF...THEN...ELSE statement, it is not considered a KNOT by MEBOW.

To each branch is added its scope weight. According to Jayaprakash, this "provides a means for recognizing branches with remote targets which when suitably accounted for, can help the complexity metric satisfy properties relating to nesting" (ibid:239). This scope weight is the weight of the subparagraph that is branched around. For a forward branch from block A to block B, the scope is the subgraph from node A + 1 to node B – 1. This represents the nodes between A and B and the branches "whose both end points are within the same set of nodes" (ibid:239). For a backwards branch from B to A, any branches that include the nodes A or B are also counted. An example from Figure 12 with a forward branch is the branch from node 4 to node 10, represented as (4,10). The scope of the branch encompasses the four branches (6,7), (6,8), (7,9), and (8,5), and the two KNOTS [(6,8), (7,9)] and [(7,9), (8,5)]. Therefore, the weight of (4,10) is equal to its raw weight plus the weight of its interior branches and KNOTS.

Figure 12 shows a program with eleven blocks, thirteen branches, and ten KNOTS. The MEBOW calculation for Figure 12 is as follows:

```
       INTEGER A,B,C
       READ 100, A,B,C
100    FORMAT (3I4)
       IF (A .GT. B)
       ______________________ 1
              GO TO 10
       ______________________ 2
       IF (B .GT. C)
       ______________________ 3
              GO TO 20
       ______________________ 4
40     PRINT 200,C
200    FORMAT (I5)
       GOTO 50
       ______________________ 5
10     IF (A .GT. C)
       ______________________ 6
              GO TO 30
       ______________________ 7
       GO TO 40
       ______________________ 8
30     PRINT 200,A
       GO TO 50
       ______________________ 9
20     PRINT 200,B
       ______________________ 10
50     STOP
       END
       ______________________ 11
```

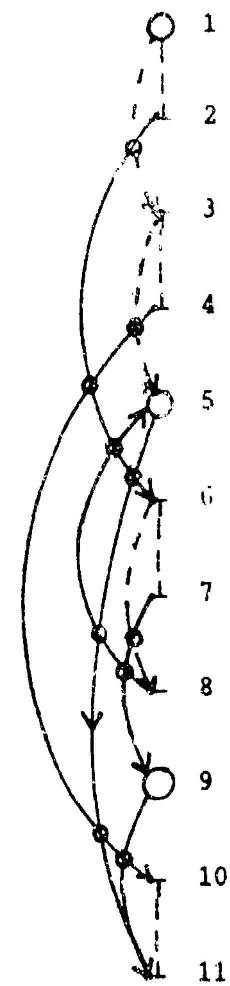

Figure 12. Example MEBOW Calculation (Jayaprakash and others, 1987:239)

Branches:

```
(1,2)   = 1   (implicit forwards branch)
(1,3)   = 1   (implicit forwards branch)
(2,6)   = 4   (explicit forwards branch = 2,
             scope covers (3,4) and (3,5) = 2)
(3,4)   = 1   (implicit forwards branch)
(3,5)   = 1   (implicit forwards branch)
```

```
(4,10)  = 27  (explicit forwards branch = 2,
              scope covers (6,7), (6,8), (7,9), (8,5) = 12 and
              KNOTS [(6,8), (7,9)] and [(7,9), (8,5)] = 13)
(5,11)  = 9   (explicit forwards branch = 2,
              scope covers (6,7), (6,8), (7,9) = 4 and
              KNOT [(6,8), (7,9)] = 3)
(6,7)   = 1   (implicit forwards branch)
(6,8)   = 1   (implicit forwards branch)
(7,9)   = 2   (explicit forwards branch)
(8,5)   = 8   (explicit backwards branch = 6,
              scope covers (6,7) and (6,8) = 2)
(9,11)  = 2   (explicit forwards branch)
(10,11) = 1   (implicit forwards branch)
```

Knots:

```
[(1,3), (2,6)]   = 5
[(2,6), (4,10)]  = 31
[(2,6), (5,11)]  = 13
[(2,6), (8,5)]   = 12
[(3,5), (4,10)]  = 28
[(4,10), (5,11)] = 36
[(4,10), (9,11)] = 29
[(5,11), (8,5)]  = 17
[(6,8), (7,9)]   = 3
[(7,9), (8,5)]   = 10

MEBOW = 243
```

This example shows MEBOW calculation, but because the program has no external interconnections, it has an INFO value of 0. An example for both MEBOW and INFO is shown in Appendix D, Calculation of Metric Value for Ada Procedure.

Some issues that have not yet been addressed are how to count compound conditions and how to count a multiway branch or CASE statement. McCabe suggests that each condition in a compound condition be counted separately (McCabe, 1983:10). For example, the statement "IF C1 AND C2 THEN" would add two to v(G) because it is equivalent to "IF C1 THEN IF C2 THEN". Critics say that this is not realistic because no matter how many conditions are considered, only one of two branches will be followed. Following this logic, the calculation of MEBOW disregards the number of conditions in a compound condition.

The CASE statement was developed to simplify multiway branch statements so that a series of nested IF...THEN...ELSE structures would not have to be created. Therefore, "It is natural to expect that a good complexity metric should assign a lower complexity value to a t-way CASE structure than its equivalent nested IF...THEN...ELSE structure" (Jayaprakash and others, 1987:242). For MEBOW, an N-way CASE statement's complexity is calculated as 2*N, instead of the N**2 complexity that would otherwise be calculated by following the MEBOW branch counting rules.

The definition of INFO states that the "fan-in of procedure A is the number of local flows into procedure A plus the number of data structures from which procedure A retrieves information" and the "fan-out of procedure A is the number of local flows from procedure A plus the number of data structures which procedure A updates" (Henry and Kafura, 1981:513). An exact description of what constitutes fan-in or fan-out is not given. For example, is an array passed into a procedure counted as 1, or as 1 for each element in the array? Is a record counted as 1, or as 1 for each field in the record that is modified in the procedure? Following the counting examples given by Kitchenham, the fan-in is considered "the number of data structures (not individual elements) the program reads from" (Kitchenham, 1988:370). The fan-out is calculated the same way. Therefore, a pointer variable or array that is input or output adds just one to fan-in or fan-out.

In languages such as Ada, it is easy to determine which procedure parameters are counted as input data flows, which are counted as output data flows, and which are counted for both. These parameters are designated "in", "out", and "in out" in the procedure declaration. This determination is more difficult in a language such as FORTRAN. In FORTRAN, a parameter or global variable is considered an output data flow if it is used on the left side (left of

the assignment operator "=") of an assignment statement, and it is considered an input data flow if it is used anywhere else. If it is used for both purposes, it is counted in both fan-in and fan-out.

To determine if a variable is global, the procedure must be parsed and each token found compared to a list of the tokens for the language being used. If the token is not a reserved word, it should be compared to the local symbol table, which is a list of those variables declared within the procedure, or compared to the procedure's parameter list. If the token is not in either of these lists, then it should be assumed that the token is a global variable. This definition may not be correct when using a language such as FORTRAN, which allows the programmer to declare variables at any point in the procedure by the use of implicit variable type declarations. Using this counting strategy, these variables would be counted incorrectly as globals which will increase the INFO value. This is incorrect, but any module that has such declarations should be monitored closely anyway because this type of declaration may be confusing to a maintainer. In Ada, a loop parameter within a FOR loop also appears to have this behavior, but this loop parameter can be easily found and added to the local symbol table during parsing because of its relation within the FOR loop parameter specification.

There can also be some difficulty in determining if a parsed token is an array or a function while counting globals. This is because in some languages such as FORTRAN, both arrays and function calls delineate their indexes and parameters with parenthesis. For example, the statement "RATE = TAXRATE(EMPLOYEE)" could either use EMPLOYEE as an index to the array TAXRATE, or EMPLOYEE could be a parameter to the function TAXRATE. If TAXRATE is not declared as a parameter or described in a COMMON block as an

array, the statement is ambiguous. The only way to determine the semantics of this statement is to parse the entire program and determine where TAXRATE is declared. Fortunately, this problem will not occur in most modern structured languages.

Following the above reasoning, one must conclude that it is not possible to make a detailed counting strategy to cover all cases. Instead, a separate counting strategy is needed for each language. This is the only way to account for the differences inherent in each language.

Threshold Value. According to Kearney, any complexity metric should have the property of normativeness (Kearney and others, 1986:1047). This means that the metric should provide an acceptable norm, or standard that specifies an allowable degree of complexity. A suitable threshold can not be determined within the scope of this research. A decision was made not to create a tool to generate the metrics, instead algorithms to implement such a tool are given in Appendix B, Algorithms for Metric Value Computation.

Threshold ranges can be calculated by creating test cases and using them with a program that calculates the metric values for programs. Taking examples of two different programs that perform the same function from sources such as the classic book The Elements of Programming Style by Kernighan and Plauger, researchers can compare the metric values to see how well they relate to the subjective opinions of structuredness and complexity offered about the programs. Another source of equivalent programs is software maintained for the Air Force in the Air Logistics Centers (ALC's). Different versions of programs that have been improved by the maintainers should be available, and the metric values can be determined for these programs and compared to the subjective judgments of the maintainability of each program. This method for obtaining programs to establish a useful threshold is preferred, as AFOTEC's task of determining

program maintainability is the primary purpose for this thesis effort. Using these programs is preferred because they will typically be larger and more complex than the academic examples, and a judgment has been made to the program's maintainability, not just if it more or less complex than another program.

Comparing the metric values to more and less maintainable programs will show more than a practical threshold value. This will also show if the two metrics should be weighted equally in the consideration of program maintainability. If one metric consistently agrees with the decisions which programs are more maintainable, then it can be weighed more than the other. Another consideration for these weights is that the test cases might suggest a change as a function of the program's characteristics. For example, programs written in one language might require different weights to better reflect maintainability than those programs written in a different language. Evaluating different classes of programs, such as avionics systems and database systems, may warrant the use of different weights for the metrics. Also, extraneous code can be added to programs, and the differences in the metric values will show the sensitivity of the indexes.

Validation of Metrics

Given this proposed method to measure maintainability, a procedure to determine how well the metrics actually measure maintainability must be developed. This is a very important consideration; it is possible to create an intuitively appealing metric that does not measure what it was intended to measure. AFOTEC is aware of the importance of validation, as is evidenced by their efforts to validate their Vol. 3 process (Lynn, 1985). An interesting discussion of validation is given in Conte:

> It is far too easy to create an attractive, intuitive model without providing data that shows that the model actually does explain the software phenomenon of interest. Attempts to validate programming models have involved collecting data via software analyzers, report forms, and interviews. Statistics have been employed to show relationships among metrics and to try to produce functions of those relationships for explanatory and predictive purposes" [Conte and others, 1986:22-23].

The validation technique suggested is for the maintaining organization to track a pilot project's history using a survey instrument. This survey instrument will record various system parameters that influence maintainability. After a certain prescribed period of maintenance has occurred, the results of surveys would be returned to AFOTEC and compared to the predicted maintainability of each program. This period of maintenance should include the first or second maintenance block changes, because by then the maintainers will have a good understanding of the system and which portions of the program are the most difficult to understand and modify. At this time, they will have enough knowledge about the software to make subjective judgments about the maintainability of each module, and a reasonable database of changed modules will be available.

Within this survey instrument a parameter of interest is the occurrence of errors. "An accepted validation technique for complexity metrics is to show the correlation of the metric to the occurrence of errors" (Henry and others, 1981:130). This is certainly a factor that drives maintenance and many experiments described in this thesis have used the occurrence of errors to reflect a program's complexity.

Another factor that should be included is the time required to modify each module, however a module may be defined. Kafura and Canning described their use of metrics to identify outliers with respect to number of errors and the amount of coding time required. In addition to time, other factors might be

considered. Other information that AFOTEC has collected in the past while validating the Vol. 3 include: program size, software language, frequency of software updates, percentage of code that changes with each update, subjective ratings of maintainability from the maintenance teams, and training time required for both new programmers and more experienced programmers (Lynn, 1985). As the following passage explains, the proposed survey tool should include all of this information, in addition to the error occurrences and the modification times for each module.

While this extra information might not be used in a correlation with metric values for the system, it may help to draw conclusions about the data received. For example, if the metrics identify one system to be more easily maintainable than a second system, yet the number of errors reported and the amount of time required to make changes on the first system is greater than the second, several factors might be involved. If the first system is maintained by novice programmers, while the second system is maintained by experienced programmers, the differences from the predicted maintainability can be explained by the experience level. Conversely, if both maintenance teams are equivalent but have differences for the error rates and time required to implement a change, then a good case can be made that the metrics are not measuring maintainability and should be modified. This extra information can be used to determine if external factors have unduly influenced the ease of program maintenance and should be taken into account during the validation process.

This type of measurement is an effort to enforce some engineering discipline on the collection and interpretation of data AFOTEC's effort to validate the Vol. 3 relied upon estimations of these factors by the project managers and senior maintenance personnel because either no detailed project data was kept by the maintenance staff, or this data was not released to

AFOTEC. A decision to collect this data must be made by both AFOTEC and the maintaining organization if a pilot study is to be made to validate these metrics.

Figure 13 shows a flow diagram of what should happen during the validation process. The left side of the diagram defines significant development milestones from the earliest point that AFOTEC is able to perform software evaluations. These development milestones are not meant to reflect the software development phases as represented in MIL-STD-2167A, but to show the major managerial changes as the program goes along.

The Operational Test and Evaluation phase occurs during the integration and test phases of software development. This phase includes two different blocks in the flow diagram. The first block depicts a baseline of the software as it is first delivered to AFOTEC for evaluation. The second block represents future evaluations of the software during the development process, which might take several years on a large system. These evaluations can be compared with the baseline to determine if the software is being made more or less maintainable because of the many changes made during the integration and test. Currently, multiple maintainability evaluations are performed on software in order to detect any problems so the developer can be directed to fix them.

The Program Management Responsibility Turnover (PMRT) represents the final version of software delivered by the developer. After PMRT the maintaining organization has maintenance control of the software. This block suggests a final post-development baseline to determine how maintainable the system that was delivered is.

The System Maintenance/Operational Support phase encompasses two blocks within the flow diagram. The first is data collection. These data reflect those

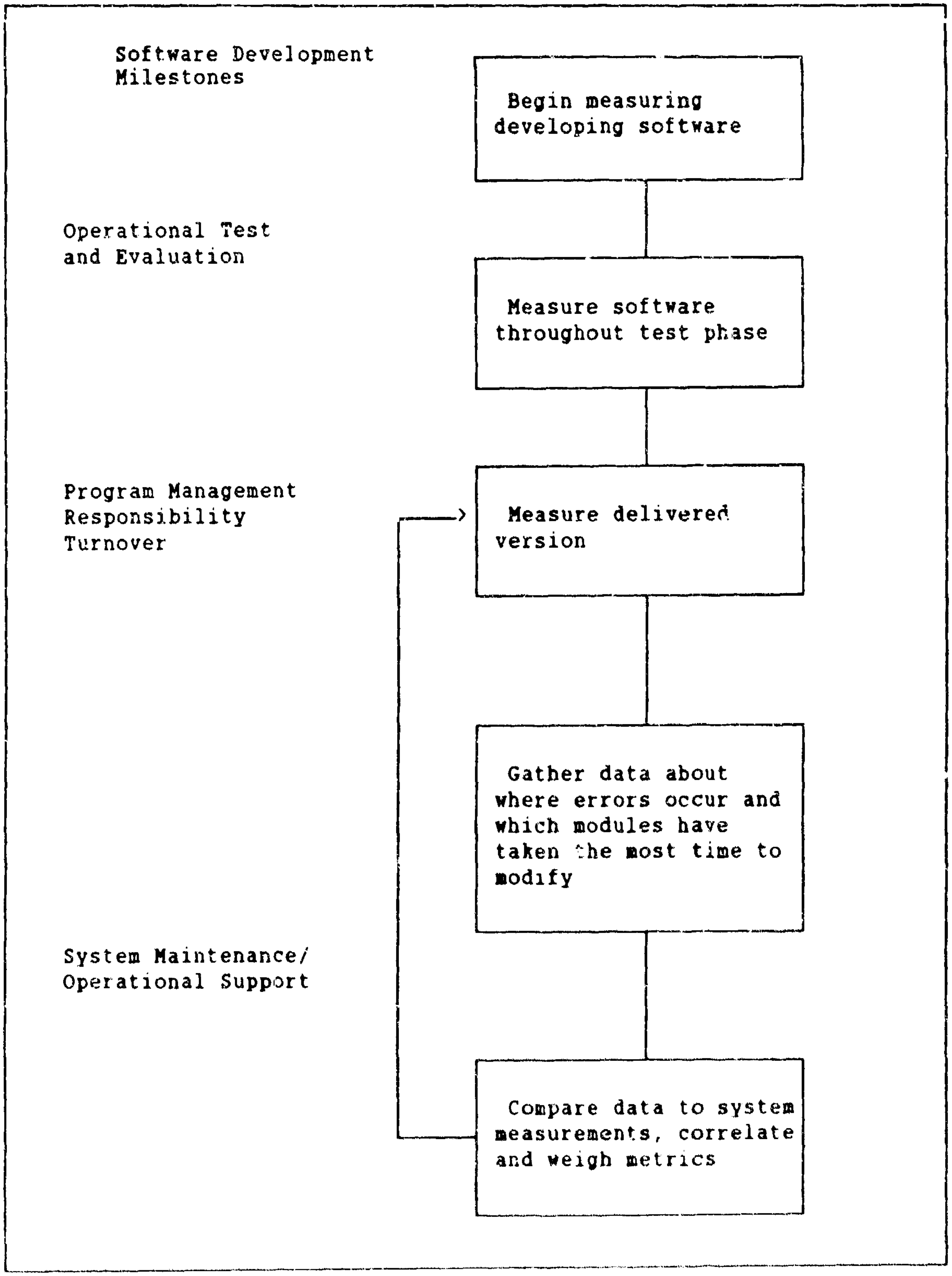

Figure 13. Example Validation Method Milestones

factors expressed above, error count and time to modify, with other factors as extra information. Once an acceptable amount of data is collected, a comparison is made with the metric values associated with the modules, and their actual error counts and modification times.

Summary

This chapter is the culmination of the work described in the previous two chapters. The use of hybrid metrics was introduced in Chapter Two, but little documentation was given to support their use. MEBOW and INFO were explained in some detail in Chapter Two, but the experiments that showed their effectiveness were not described. Chapter Three demonstrated how to determine if a metric measures maintainability and two metrics were exhibited as being able to measure maintainability effectively if paired together.

This chapter provided a lengthy description of the usefulness of hybrid metrics, along with descriptions of experiments indicating the use of hybrid metrics as a valuable measurement technique. Then supporting evidence was given that MEBOW and INFO are both useful metrics and have been used to measure complexity successfully. Metric calculation issues were discussed and examples were given showing the application of the described counting strategies.

While the use of hybrid metrics and the two metrics described have been documented, it is understood that they have not been used together. Therefore, important information about the metrics such as their sensitivity, useful threshold values, and if one metric better reflects maintainability for the types of programs these will be used to measure, have not been determined. This is a reason to emphasize the importance of metric validation.

V. Conclusions and Recommendations

Introduction

This research has involved a survey of metrics, a definition of criteria to determine which metrics measure maintainability, and an in-depth look at two metrics which meet the most of these criteria. A determination will be made now whether using these metrics will actually solve he problems that were given in Chapter One to be resolved. This chapter explains what was accomplished and how the problem was met. Then the limitations of the solution will be elaborated. Finally, some recommendations for further research are examined.

Conclusions

A framework for the automated evaluation of software maintainability was developed. A set of criteria to determine which automatable complexity metrics better reflect maintainability than others was defined. The metrics discussed were compared with each criteria and the metrics that had the most complete coverage in each of the three criteria groups was determined. A combination of the two metrics MEBOW and INFO was determined to have the best coverage overall of the criteria.

After these two metrics were selected for use, their implementation was studied. Algorithms to calculate the two metrics were developed, although a tool to generate the metrics was not created. A method to determine useful threshold values for these metrics was explained. Sources for test cases to determine these thresholds were given. A procedure to validate the use of these metrics to measure maintainability was developed. This procedure specified what data must be collected, and when it should be collected.

The following section examines two issues: how the given problem was solved, and the limitations and benefits of automated metrics are presented (or how metrics can be helpful once the limited information they yield is understood).

<u>How the Problem was Solved</u>. The problem was to develop software maintainability metrics to be incorporated into the AFOTEC Vol. 3. Constraints on the metrics researched were that they had to fit into the scope of the seven characteristics of maintainability explained in Chapter One. These metrics were to measure aspects of maintainability that the Vol. 3 does not, and they must be automatable. The following discussion presents each of these issues.

These metrics were to be incorporated into the Vol. 3, and applied within the seven maintainability characteristics. This was only partially fulfilled. When this research began, the metrics were to be incorporated into the Vol. 3 as questions, with equal weight as the other questions. As more information about metrics was acquired, though, a decision was made not to include these metrics within the overview of the Vol. 3, but to use them in a separate maintainability evaluation whose results could support the Vol. 3. Also, separate metrics to reflect each of the seven maintainability characteristics were not discovered. Simplicity is the only characteristic that is measured by the proposed metrics.

These metrics were to measure aspects of maintainability that the Vol. 3 does not. MEBOW reflects complexity issues such as the levels of nesting in control structures, and the scope of branches. These are addressed within the Vol. 3, but are not measured in the same way that MEBOW does. The Vol. 3 asks the evaluator for a subjective estimate of how complex is a module's nesting, while MEBOW objectively calculates the complexity caused by nesting. INFO reflects the complexity of the data connections between modules, while the Vol. 3 subjectively measures the number of global variables used, and how well

the input and output parameters to a module are described. Therefore, MEBOW and INFO measure some different aspects of maintainability than the Vol. 3 does. These metrics can be automated and used to evaluate all of a program's source code, instead of just a fraction of it.

Each aspect of the problem has been considered, and the constraints have been met. This use of an automated tool that calculates MEBOW and INFO to measure the maintainability of software is a solution for the problem given. The aspect of this problem that was not sufficiently answered is the automated use of metrics that measure qualities supporting the six characteristics other than simplicity.

The Limitations and Benefits of Metrics. The metrics presented can be used to reflect maintainability, if they are used correctly and their limitations are understood. These metrics can be used to gather data, but the interpretation of this data must be made with a clear understanding of what the data mean. Rodriguez and Tsai state in their conclusion, "The final conjecture states that the metrics should not be accepted as axioms. They give information, but that information has to be interpreted in the context of the particular system being measured" (Rodriguez and Tsai, 1986:368).

Metric analyses are useful only to compare "apples with apples", which is a reason that a well-defined counting strategy is needed (Conte and others, 1986:27). Someone comparing two different sets of data should have confidence that they were both counted in a consistent manner. These software metrics require calibration from historical data gathered in a specific environment to establish appropriate weights and threshold values (ibid). The suggested metrics have been shown to reflect the complexity of modules written in procedural languages, but no evidence supports their use with different paradigms. An

application of MEBOW to modules written in languages such as LISP, PROLOG, or Smalltalk might not be practical. Any comparison of measured values from modules written in one of these languages and other modules written in a procedural language such as Ada might not reflect their relative complexity.

While software metric results can assist the decision-making process of software development and testing personnel, they cannot replace this process (ibid). A module that rates below the threshold value may still be more difficult to maintain than one that scores above the threshold, depending on maintainability factors that are not measured by the proposed metrics. For example, a well-commented module that is complex may be more easily maintained than a less complex module that has no comments. This factor is not reflected by the automated metrics. That is why the metrics should be used in an advisory capacity, as Harrison and Cook suggest, "On the practical side, our study suggests that software project managers can use software complexity measures as a tool in identifying the few subprograms most likely to contain the majority of errors, and hence can allocate their testing resources more efficiently" (Harrison and Cook, 1987:214).

Along these same lines, how the software development managers use the metrics should be limited. According to Conte, "software metrics and models are intended to be used to manage products, not for evaluating the performance of technical staff" (Conte and others, 1986:27-28). If the programmers understand that their performance is being measured, they will quickly find ways to realize improved metric results, even if this does not improve the maintainability of the program.

Another limitation is that the difficulty and cost of computing metrics may be high (ibid). This is a problem with the use of the Vol. 3 evaluation

technique. The automation of MEBOW and INFO should lessen the cost of measuring these metrics.

Several benefits of using these automated metrics have been briefly described. One benefit is that modules that are most likely to contain errors will be identified, and greater testing resources can be allocated to those modules. Also, those modules that are too complicated are recognized, and they can be further decomposed to less complex, more maintainable modules. Côté expresses an interesting analogy for the use of metrics by asserting, "metrics can greatly help in depicting the features and layouts embedded in thousands of lines of code, in much the same way that gauges and dials give a nuclear plant operator an idea of what is going on inside a reactor" (Côté and others, 1988:121).

These sections have shown that the automated calculation of MEBOW and INFO will resolve the problems this research has attempted to solve. The limitations inherent with the use of automated metrics have been described, along with the benefits from their use. Metrics have an important application, but should not be used out of their limited context.

Recommendations

The use of a hybrid control and data structure metric appears to answer AFOTEC's needs. Before these metrics should be used, though, certain issues must be considered. Recommendations to resolve these issues are explained.

The first recommendation is that a tool that measures both MEBOW and INFO must be built. Although either metric value can be computed manually, this process is difficult and time-consuming. This manual computation would also violate one of the constraints given, that no extra work be given to the

evaluators. An implementation of these metrics in an automated tool is a necessary first step for the use of these metrics.

Until some type of study has shown that this hybrid metric reliably reflects maintainability, its use should be considered advisory. A pilot study following the validation method presented in Chapter Four will relate the metric values to maintainers' subjective ideas of which modules were more maintainable. Once this study has been accomplished, a determination can be made if the metrics actually reflect maintainability. If the study suggests they do reflect maintainability, the metrics' results can be used in the same manner as the Vol. 3 evaluation results. If the study suggests the metrics do not measure maintainability, perhaps some weighting of the metrics can be used that will better reflect maintainability, or a different class of metric can be included. While data for this study is being accumulated, these metrics can be compared to the Vol. 3 results and other subjective measurements of maintainability.

The amount of data that will have to be collected by maintainers for this pilot study may cause objections from the maintainers. They may resent the amount of time and effort required for a study that will not immediately support their office. The importance of collecting this data must be emphasized to the maintainers, as well as the positive impact to the evaluation of maintainability of future software systems.

A recommendation for further study is for someone to complete the validation pilot study. This is most likely an effort that AFOTEC will have to provide for itself, as the data collection may take some time. Once this data collection has been accomplished, following the guidelines presented in the previous chapter, a comparison should be made with the metrics' maintainability predictions and the collected maintainability data.

Another important area is to discover software metrics that reflect characteristics other than just complexity. For example, Harrison and Cook presented a measure of "documentation" that they use to reflect how self-descriptive a module is (Harrison and Cook, 1987:215). This measure is just a ratio of the number of comments to the number of total lines of the module. But perhaps other descriptiveness metrics are available which cannot be so easily thwarted. Possibly some metric that reflects how modular the software is can be developed. These types of metrics could be used along with the complexity metrics already suggested and would broaden the metric coverage to determine software maintainability.

<u>Summary</u>

The use of metrics to measure software's complexity and maintainability shows much promise, even if the initial fascination with some metrics such as Halstead's Software Science and McCabe's cyclomatic complexity has worn off. This is well described by the conclusions of Kafura and Canning:

> Even if software metrics had no other use their proven ability to identify the most error-prone components would be of tangible value to software developers. This tangible value is particularly evident if the structure metrics can be used to identify the most error prone components since this would permit the system to be redesigned so as to avoid components of this type altogether. Furthermore, information on error-prone components would allow the testing or code review processes to be concentrated on these components [Kafura and Canning, 1985:381].

Appendix A: Justification for Metric Complexity Criteria Ratings

In Chapter Three, Figure 9 shows a matrix of metrics vs. the metric selection criteria developed in that chapter. Within the matrix, marks are shown that indicate if the metric meets the metric selection criteria, and the level of agreement if it does so. This appendix explains the reasoning behind the agreement indications. The metrics are listed in the same order as they are shown in Figure 9, and remarks are given for each criteria.

LOC

Clear and Unambiguous: As the example in Figure 1 shows, LOC calculation is ambiguous without a definite counting strategy.

Intuitive: A longer program is likely to be more difficult to maintain than a shorter one.

Language Independent: Each language needs a different counting strategy.

Prescriptive: If a module is significantly longer than others in the same program, it is a candidate for further decomposition. This does not give any indication how the module should be broken up, though.

Robustness: Making a program shorter by breaking it up into modules should lessen its complexity. This will be reflected by the metric. But as a counter-example, if all the comments are taken out of the module, it will be shorter, but more complex.

Accurately Reflect Control Flow: NA

Ranking Basic Control Structures: NA

Nesting and Compound Conditions: NA

Accurately Reflect Data Flow: NA

Indicates Data Amount: NA

Shows Data Use: NA

Reflects Inter-Module Data Links: NA

N

Clear and Unambiguous: Chapter Two explains that some tokens can be both operators and operands, which complicates the metric calculation.

Intuitive: As programs are composed of operands and operators, a program with more operators and operands is likely to be more difficult to understand than one with fewer operators and operands.

Language Independent: In languages such as LISP, the difference between operators and operands is not clear.

Prescriptive: A module that has a larger number of operators and operands should be decomposed into shorter modules, but this gives no suggestion how to accomplish the decomposition.

Robustness: If a module is changed, operators will either be added or deleted. This will reflect that a change occurred.

Accurately Reflect Control Flow: NA

Ranking Basic Control Structures: NA

Nesting and Compound Conditions: NA

Accurately Reflect Data Flow: NA

Indicates Data Amount: While the total number of operands might be considered a reflection of the amount of data used in a program, N includes both operators and operands and does not show just the data.

Shows Data Use: NA

Reflects Inter-Module Data Links: NA

Span

Clear and Unambiguous: The number of lines between two variable references is not difficult to count.

Intuitive: The fewer the number of lines between variable references, the more likely a maintainer will be able to understand a variable's usage.

Language Independent: This can be used with any language that has variables and lines between them.

Prescriptive: If a variable has a large span, it is possible that the module is too large and should be decomposed.

Robustness: A change in the number of lines between two references will be reflected, as well as any added variable references will change the span for that variable.

Accurately Reflect Control Flow: NA

Ranking Basic Control Structures: NA

Nesting and Compound Conditions: NA

Accurately Reflect Data Flow: NA

Indicates Data Amount: NA

Shows Data Use: This reflects how data is used within a module to the extent that it shows locality of variable references.

Reflects Inter-Module Data Links: NA

INFO

Clear and Unambiguous: The definition presented by Henry (Henry and Kafura, 1981) is straightforward. Others have not been able to calculate these values. For example, Harrison and Cook did not use (fan-in * fan-out) ^2 in their calculations (Harrison and Cook, 1987). This does not suggest that the INFO calculation is ambiguous, instead, it reflects their inability to separate their data into fan-in and fan-out. This is not a problem in the interpretation of a counting strategy, such as LOC.

Intuitive: The greater the number of connections to other modules, the greater the possible impact of any change and the higher the complexity.

Language Independent: Any language that can be broken into modules can have the data links measured.

Prescriptive: Henry and Kafura used INFO to show which modules in the Unix kernel were data "choke-points" (Henry and Kafura, 1981:517). This gives an indication of the effect that modifying a module will have on the other modules in a system.

Robustness: If a change to the number of data items referenced or modified is made, this will reflect the change. This will not reflect a change to how the data is used, or any change in the module control flow.

Accurately Reflect Control Flow: NA

Ranking Basic Control Structures: NA

Nesting and Compound Conditions: NA

Accurately Reflect Data Flow: This will reflect the inter-module data flow, but not the intra-module data flow.

Indicates Data Amount: This reflects the parameter and global data flows into and out of a module.

Shows Data Use: This shows the amount of data, but not its use, in a module.

Reflects Inter-Module Data Links: This reflects the parameter and global data flows into and out of a module.

v(G)

Clear and Unambiguous: Different counting strategies have been introduced for control structures such as case statements.

Intuitive: The greater the number of branches in a module, the more difficult it will be to understand.

Language Independent: This operates on a directed graph representation of a program, so it is independent.

Prescriptive: This tells when a program has become too complex. By viewing the control flow graph, a determination can be made how sections of code can be separated into a different module without adversely affecting the structure of the original module.

Robustness: A change in a control structure may or may not be reflected. A rearrangement of a module that contains the same number of branches will not have a different v(G) value. Any change to sequential statements will not be reflected, nor will any change to the data flows.

Accurately Reflect Control Flow: This does represent the control flow of a module.

Ranking Basic Control Structures: A series of sequential statements is presented as less complex than a branch. An IF...THEN...ELSE branch is shown as more complex than an IF...THEN branch. An iteration construct is more complex than a sequential statement.

Nesting and Compound Conditions: Arguments were given in Chapter Two saying that this does not reflect nesting or compound conditions.

Accurately Reflect Data Flow: NA

Indicates Data Amount: NA

Shows Data Use: NA

Reflects Inter-Module Data Links: NA

Knot

Clear and Unambiguous: A crossing of control paths is easily understandable. If the language in use allows multiple statements on one line, though, some difficulty in determining if a knot occurs may arise.

Intuitive: If structured programming is considered to be a useful paradigm, then any reflection of unstructuredness will show added complexity.

Language Independent: The problem of calculating the knot count with a language that allows multiple statements on a line arises.

Prescriptive: According to Woodward, (Woodward and others, 1983) if a module with knots is rewritten to have fewer knots, it will be less complex. The problem is how to rearrange the code to have fewer knots.

Robustness: A change in a module's control structure will be reflected. But any change to sequential code or data flow will not be reflected.

Accurately Reflect Control Flow: This does not represent the program's underlying control flow, instead, it reflects the unstructuredness of the source code text.

Ranking Basic Control Structures: No ranking is given.

Nesting and Compound Conditions: This does reflect nesting and any branches out of nested control structures.

Accurately Reflect Data Flow: NA

Indicates Data Amount: NA

Shows Data Use: NA

Reflects Inter-Module Data Links: NA

MEBOW

Clear and Unambiguous: This is slightly more complex to understand than either v(G) or Knot, but is precisely defined.

Intuitive: The more complex the control structures are in a module, the more complex the module is.

Language Independent: This operates on a directed graph representation of a program.

Prescriptive: This tells when a program has become too complex. By viewing the control flow graph, a determination can be made how sections of code can be separated into a different module without adversely affecting the structure of the original module.

Robustness: To a greater extent than either v(G) or Knot, this will reflect any changes in a module's control flow. But this has their same limitation that it does not reflect data flow or the amount of sequential code.

Accurately Reflect Control Flow: This reflects the control flow as well as v(G), and shows unstructuredness as well as Knot.

Ranking Basic Control Structures: This reflects the ordering: sequential statements < condition statements < iteration statements.

Nesting and Compound Conditions: This reflects nesting because of the scope component in each branch's value.

Accurately Reflect Data Flow: NA

Indicates Data Amount: NA

Shows Data Use: NA

Reflects Inter-Module Data Links: NA

<u>E</u>

Clear and Unambiguous: This has the same counting ambiguities as <u>N</u>.

Intuitive: Studying the total and unique operators and operands is understandable. The weighting factors in the E value lessen its easy understanding.

Language Independent: This has the same problem as <u>N</u>.

Prescriptive: A higher value will suggest that the program needs to be decomposed, but does not give any guidelines how to implement this decomposition.

Robustness: Changing the number of operators or operands will make a difference in the metric value, especially if the operator or operand has not been used yet in the module.

Accurately Reflect Control Flow: NA

Ranking Basic Control Structures: NA

Nesting and Compound Conditions: NA

Accurately Reflect Data Flow: NA

Indicates Data Amount: NA

Shows Data Use: NA

Reflects Inter-Module Data Links: NA

$v(G)$, η_1

Clear and Unambiguous: The calculation of v(G) was explained above. Counting the number of unique operators may be difficult, as the <u>N</u> section reflects.

Intuitive: Adding operators to the branches should show more about the modules's complexity than either will separately.

Language Independent: While v(G) is language independent, the operator count is not.

Prescriptive: This comes from the description of v(G) prescriptiveness.

Robustness: A change in either the number of branches or the number of unique operators will be reflected by the metric. Any change in data will not be reflected.

Accurately Reflect Control Flow: This will reflect control flow as well as v(G).

Ranking Basic Control Structures: This ranks structures as well as v(G).

Nesting and Compound Conditions: This has v(G)'s limitations in reflecting nesting.

Accurately Reflect Data Flow: NA

Indicates Data Amount: NA

Shows Data Use: NA

Reflects Inter-Module Data Links: NA

C

Clear and Unambiguous: This is a very complex metric to calculate.

Intuitive: Using both a data structure metric with a control flow metric gains the benefits explained for a hybrid metric.

Language Independent: This does not reflect any particular language, as long as a procedural language is used.

Prescriptive: A high data flow or control flow component will suggest that the module needs to be decomposed, but doesn't explain how to best decompose the module.

Robustness: Any change of "locally exposed" data references will be reflected, and changes in the control flow will be reflected.

Accurately Reflect Control Flow: This reflects the number of branches in a module.

Ranking Basic Control Structures: This ranks sequential statements as less complex than conditions.

Nesting and Compound Conditions: This reflects nesting, and reflects the use of data within nested statements.

Accurately Reflect Data Flow: This shows the use of data and the amount of data that are "locally exposed".

Indicates Data Amount: The data flow factor reflects the number of variables used.

Shows Data Use: The use of data within a module is reflected within each node of the module.

Reflects Inter-Module Data Links: This is an intra-module metric.

Appendix B: Algorithms for Metric Value Computation

This section explains algorithms for calculating the metric values for both MEBOW and INFO. The INFO calculations are given first, then the MEBOW algorithms are explained. Each section will have an explanation, followed by a pseudocode representation of the algorithm.

Information Flow Calculation. This calculation requires a list of the reserved words for the language the module under evaluation is written in. This algorithm assumes that a parser is available that can return tokens for the language being used, along with a list of parameters from the module calling statement. The algorithm goes through the module, looking for valid identifiers, and compares each identifier to this list of reserved words. If the identifier is in the list, it is discarded and the next identifier is evaluated. When the module's parameter declarations are evaluated, any identifier used as an input variable is added to the input list and any identifier used as an output variable is added to the output list. Any variables declared locally are placed in the reserved word list so they will not be counted as input or output data flows.

If the identifier is not in the list, depending on how the identifier is used, it is compared to either a list of input identifiers or a list of output identifiers. If the identifier is on the left side of an assignment, it is compared to the output identifiers. If it is used in any other expression, it is compared to the list of input identifiers. If the identifier is not in the appropriate list, it is added to that list. This operation continues until the end of the module is reached.

```
-- This section adds the module's parameters to the input and output
-- identifier lists.

REPEAT

    IF the identifier is an input parameter
      THEN

          ADD the identifier to the list of input identifiers

    ELSE

          ADD the identifier to the list of output identifiers

    ENDIF

UNTIL no more parameters are found


-- This section adds the module's locally declared variables to the list
-- of reserved words so they will not be counted for the INFO value

REPEAT

    IF a valid variable declaration is found
      THEN

          ADD the identifier to the list of reserved words

    ENDIF

UNTIL the beginning of the program body is found


-- This section looks for identifiers within the body of the module and
-- adds these identifiers to the input and output identifier lists.

REPEAT

    INPUT an identifier
    COMPARE the identifier to a list of reserved words
    IF the identifier is not in this list
      THEN

          IF the identifier is on the left side of an assignment
            THEN

                COMPARE the identifier to a list of output identifiers
                IF the identifier is not in this list
                  THEN

                      ADD the identifier to the list of output identifiers
```

```
            ENDIF

        ELSE        (the identifier is not being assigned a value)

            COMPARE the identifier to a list of input identifiers
            IF the identifier is not in this list
              THEN

                  ADD the identifier to the list of input identifiers

            ENDIF

        ENDIF

    ENDIF

UNTIL the end of module is reached

CALCULATE INFO as (the number of input identifiers *
                   the number of output identifiers) ^ 2
```

<u>MEBOW Calculation</u>. Each statement within the module that is either a branch or the target of a branch is kept track of by its line number. A list of branch/target pairs is kept, along with a determination of the type of branch. The types of branches are implicit or explicit, and backwards or forwards, each having a different value for MEBOW calculations. Once all of the branch/target pairs have been determined for the module, each branch's value is calculated.

Each branch is assigned the weight of its type. For example, if a branch from line 13 to line 17 is implicit, the branch (13,17) is assigned 1, or the explicit branch (15,8) is assigned 6. Then, the scope of each branch is calculated. A list of branches that are within the scope of each branch is determined, along with any knots within that scope. If a branch has no other branches or knots within its scope, it is flagged as completed. This list that contains the branches and knots within the scope of a branch is a list of those values the branch depends on, or its dependency list.

The value for each branch is calculated as the sum of its weight and the sum of the values for the branches and knots in its dependency list. If each of these branches and knots has its completed flag set, then the current branch's value can be calculated and its completed flag set. If any of these branches or knots is not completed, the current branch is bypassed for later calculation. This is an iterative process, which continues through the list of branches until all dependencies have been completed. The sum of all the branch's values is the returned MEBOW value.

A case statement will be treated somewhat differently. A list of the line numbers for its selections will be kept, along with their number. The calculation of its value is 2 * the number of selections, added to the value of each selection. These values are calculated the same as any other statements, and dependency lists are kept for case statements, also. Once each selection's dependencies have been evaluated, the case statement's value can be calculated.

```
-- This creates the list of branches

REPEAT

      IF the statement is a branch
        THEN

            ADD the line number to the list of branches, along with a
                  determination of what type of branch it is

      ELSE

            IF the statement is the target of a branch
              THEN

                  ADD the line number to its corresponding branch, to
                        create a (TO, FROM) representation

            ENDIF

      ENDIF
```

```
UNTIL the last statement is input

-- This generates the list of dependencies for each branch and determines
-- if any knots have occurred.  Any knots are kept in a separate list,
-- and their dependencies are also generated.

REPEAT

      COMPARE the line numbers for a branch to each other branch
      IF overlap of line numbers exists
        THEN

            IF the criterion for a knot exists
              THEN

                  ADD the knot to the list of knots

            ENDIF

            ADD the dependency to the current branch
            SET the completion flag to false

      ELSE

            SET the completion flag to true

      ENDIF

UNTIL all branches and knots have had their dependencies evaluated


-- This goes through the lists of branches and knots and determines
-- which have enough information available to calculate their value.
-- If they depend on the value of a branch or knot that is not yet
-- known, then pass to the next branch or knot and try to calculate
-- its value.

REPEAT

      IF the completion flag of a branch or knot is false
        THEN

            CHECK the completion flags for each branch and knot in its
                  dependency list
            IF all of the completion flags are true
              THEN

                  ADD the values for each branch and knot in the
                        dependency list
                  SET the completion flag to true

            ENDIF
```

```
    ENDIF
UNTIL all completion flags are set

-- The MEBOW value is the sum of the values for the branches and the knots
REPEAT
    ADD the value of the current branch or knot to the SUM
UNTIL all branches and knots have been added

-- The MEBOW value is now known.
```

Appendix C: Empirical Support for Hybrid Metrics

Empirical Evidence.

In Chapter Four, descriptions of four studies that use hybrid metrics were given. The data and results that came from these studies are presented here in more detail. This section shows the data for three of the four studies.

Kafura and Canning's Study. Kafura and Canning's research identified 32 components that were extreme outliers, or the most error-prone components, from within the 170 components they studied. They analyzed these 32 components with their ten metrics. They found that 28/32 extreme outliers were identified by at least one of the ten metrics (Kafura and Canning, 1985:382). The best result that any single metric had was 20/32. This metric was the INFO-LOC hybrid measure. Figure 14 shows a matrix of the metrics and the extreme outlier components they identified.

This analysis was also performed on all the outliers, not just the extreme outliers. The number of components in this category totaled 85. INFO-LOC identified 42/85 outliers. This the best result of any metric used. This metric also identified the second fewest number of non-outliers as outliers. All of the metrics incorrectly identified some components as outliers, but INFO-LOC's percentage of correctly identified outliers to total outliers presented was 42/63, which is only beaten by LOC's yield of 41/59. While other metrics identified fewer non-outliers as outliers, they also did not recognize as many of the correct outliers.

Harrison and Cook's Study. Harrison and Cook's results showed that their hybrid metric was able to identify the most error-prone modules better than any other metric used. The results in Figure 15 show that MMC as a

Software Metric	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	Metric Total
LOC	X	X	X		X		X	X		X		X		X				X		X	X			X	X	X			X	X		X	18
EFFORT		X			X				X																	X						X	5
CYCLO	X	X			X							X		X						X					X	X				X		X	10
INFO	X	X	X	X		X	X	X		X	X	X	X	X				X	X		X			X	X	X			X				19
INVOKE	X	X	X	X	X		X	X									X	X	X	X		X		X			X		X	X	X		17
REVIEW	X	X				X	X					X									X			X	X								8
STABILITY		X		X		X	X	X			X	X	X					X	X					X	X								12
INFO-LOC	X	X	X	X	X	X	X	X		X	X	X	X	X				X	X		X			X	X	X			X				20
REV-LOC	X	X					X			X		X		X				X		X	X			X	X	X							12
STAB-LOC	X	X		X							X		X	X			X	X	X			X		X	X						X		13
Component Totals	8	10	4	5	5	4	7	5	1	4	4	7	4	6	0	0	2	7	5	4	5	2	0	8	8	6	1	0	4	3	2	3	

Figure 14. Identification of Extreme Outlier Error Components (Kafura and Canning, 1985:382)

Metric	BUGS	MMC	HNP	HNK	DSL	VG	E	PRC
MMC	.82	---	.90	.77	.67	.91	.80	.79
HNP	.75	.90	---	.84	.70	.95	.87	.92
HNK	.62	.77	.84	---	.77	.89	.98	.91
DSL	.76	.67	.70	.77	---	.77	.81	.70
VG	.73	.91	.95	.89	.77	---	.93	.92
E	.69	.80	.87	.98	.81	.93	---	.93
PRC	.64	.79	.92	.91	.70	.92	.93	---

Pearson Product Moment coefficients of correlation for each of the metrics vs. the number of errors and the other metrics. MMC: the new metric; HNP: Hall and Preiser's Metric; HNK: Henry and Kafura's Metric; DSL: Delivered Source Lines; VG: McCabe's Metric; E: Halstead's Effort; PRC: number of procedures.

Figure 15. Results of Harrison and Cook's Study (Harrison and Cook, 1987:217)

hybrid of different types of metrics outperforms its component parts. They wrote, "as can be seen, the MMC metric performed significantly better than any of the other metrics examined" (Harrison and Cook, 1987:217). The MMC metric had a .82 correlation with the number of errors found in the modules tested. This metric was "based loosely" on the Hall and Preiser Combined Network Complexity metric and Henry and Kafura's INFO, with a microcomplexity metric of v(G) (ibid:215-216).

Ramamurthy and Melton's Study. Ramamurthy and Melton did not perform a statistical analysis of the quality of their weighted metrics. Instead, they showed comparisons of the unweighted and weighted Software Science and v(G) metrics against 24 pairs of test programs and program segments. In three tables, each pair is shown with the first program as the more complex of the two. No justification how the first program was identified as more complex was given.

Their first table showed six programs with the same Software Science values but different v(G). Their weighted metrics identified the first program of the test pair as more complex in all cases. Their second table showed eight programs with the same v(G) but different Software Science values. In all cases, the weighted effort showed the first program in the test pair as more complex. In one case, the weighted length and volume measures incorrectly identified the less complex program, although the weighted effort for the same program was correct. Their third table showed ten programs with different Software Science and v(G) values. The weighted length and volume metrics correctly identified all ten programs. The weighted effort metric incorrectly identified one program. Overall, the weighted metrics correctly identified 23/24 programs as being more complex than the less complex of the test pair. The Software Science metrics correctly

identified 16/24, and v(G) correctly identified 15/24. This suggests that a hybrid measure identifies complexity better than a single metric can (Ramamurthy and Melton, 1986:312).

Appendix D: Calculation of Metric Value for an Ada Procedure

This appendix shows the calculation of both metrics for an Ada procedure. This procedure was taken from a program that calculates v(G) for Ada programs, and is in the public domain. First the MEBOW calculation is presented, then a list of the input and output variables will be given with a calculation of information flow.

The MEBOW calculation for this procedure will not be presented in the same format as Figure 12. The flow graph for this example is complicated and will not add to the comprehensibility of the example. Instead, Figure 16 shows MEBOW values for five basic control structures. These structures are labeled in the example. Their MEBOW values, which are their basic values added to their scope, are presented after the example.

The first line of a control structure branch used in the MEBOW calculation is labeled with a designator for reference during the MEBOW value calculation. This designation is "C" for a case statement, "I" for an if statement, and "W" for a while loop. These designators are numbered sequentially, so "I3" refers to the third occurrence of an if statement.

Figure 16 shows the MEBOW values for the if statements and while statements used, but does not refer to the case statements. Each case statement's MEBOW value is twice the number of branches, which are its enclosed "when" statements. To this value the scope of each enclosed branch is added, to give the MEBOW value for the branch. For example, the case statement C4 has three branches, and encloses C5, C6, and I5 within its scope. Therefore, its value is (2 * 3) + C5 + C6 + I5 = 29.

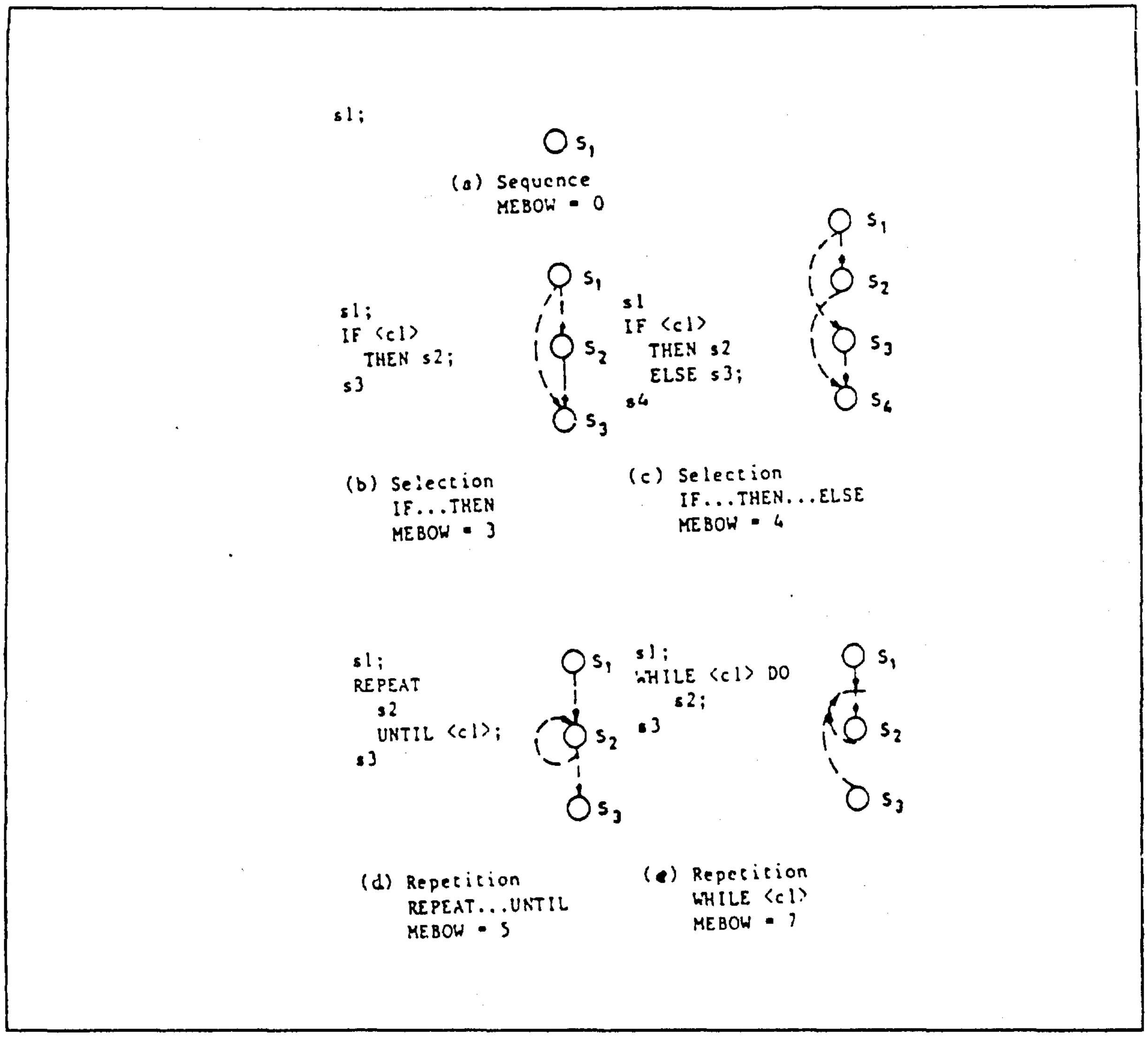

Figure 16. MEBOW Basic Control Constructs (Jayaprakash and others, 1987:240)

```
procedure Scan_Numeric_Literal;    --| Scans numbers

 --| Requires
 --|
 --| This subprogram requires an opened source file, and the
 --| Universal Arithmetic package to handle conversions.
 --|

 --| Effects
 --|
 --| This subprogram scans the rest of the numeric literal and converts
 --| it to internal universal number format.
 --|

 --| Modifies
 --|
 --| CST
 --|


 procedure Scan_Numeric_Literal is

 --| Overview
 --|
 --| Note the following LRM Sections:
 --|     LRM Section 2.4    - Numeric Literals
 --|     LRM Section 2.4.1 - Decimal Literals
 --|     LRM Section 2.4.1 - Notes
 --|     LRM Section 2.4.2 - Based Literals
 --|     LRM Section 2.10   - Allowed Replacements of Characters
 --|

 ------------------------------------------------------------------
 -- Declarations for Scan_Numeric_Literal
 ------------------------------------------------------------------

 Based_Literal_Delimiter : character;
     --| holds value of first based_literal delimeter:
     --| ASCII.COLON (':') or ASCII.SHARP ('#');
     --| so the second one can be checked to be identical.

 Base_Being_Used : GC.ParserInteger;
     --| base value to be passed to Scan_Based_Literal.

 ------------------------------------------------------------------

 begin

     CST.gram_sym_val := PT.NumericTokenValue;

     Work_String_Length := 0;
     -- also used by sub-scanners called from this subprogram.
```

```
           -- Scan first field
           Scan_Integer;

           -- Now, scan rest of literal dependent on what Next_char is
C1         case Next_Char is

               -- have a decimal_literal
               when '.' =>
I1                 if (Look_Ahead(1) = '.') then
                       -- next token is a range double delimiter.
                       -- finished with numeric_literal.
                       Seen_Radix_Point := false;  -- have an integer_literal
                       -- already set_up for next scanner,
                       -- no call to Get_Next_Char.
                   else
                       Seen_Radix_Point := true;
                       Add_Next_Char_To_Source_Rep;
                       Get_Next_Char;
C2                     case Next_Char is
                           when Digit =>
                               Scan_Integer;
                               -- check and flag multiple radix points
W1                             while (Next_Char = '.') and then
                                   (Look_Ahead(1) in digit) loop
                                       LEM.Output_Message
                                           ( Current_Line
                                           , Current_Column
                                           , LEM.Too_Many_Radix_Points);
                                       Add_Next_Char_To_Source_Rep;
                                       Get_Next_Char;
                                       Scan_Integer;
                               end loop;
                           when ASCII.UNDERLINE =>        -- '_'
                               -- flag illegal leading under line
                               LEM.Output_Message(
                                   Current_Line
                                   , Current_Column
                                   , LEM.Leading_Underline);
                               Scan_Integer;
                               -- not flagging an integer consisting of a
                               -- single underline as a trailing radix
                               -- point case.  Check and flag multiple radix
                               -- points.
W2                             while (Next_Char = '.') and then
                                   (Look_Ahead(1) in digit) loop
                                   LEM.Output_Message(
                                       Current_Line
                                       , Current_Column
                                       , LEM.Too_Many_Radix_Points);
                                   Add_Next_Char_To_Source_Rep;
                                   Get_Next_Char;
                                   Scan_Integer;
                               end loop;
```

```
                    when others =>
                        -- flag trailing radix point as an error
                        LEM.Output_Message(
                            Current_Line
                          , Current_Column
                          , LEM.Digit_Needed_After_Radix_Point);
                end case;

                Scan_Exponent;  -- check for and process exponent

            end if;

        -- have a based_literal
        when ASCII.SHARP |      -- '#'
             ASCII.COLON =>     -- ':'
I2          if (Next_Char = ASCII.COLON) and (Look_Ahead(1) = '=') then
                -- next token is an assignment compound delimiter
                -- finished with numeric literal.
                Seen_Radix_Point := false;  -- have an integer literal
                -- already set up for next scanner, no call to
                -- Get_Next_Char.
            else
                Based_Literal_Delimiter := Next_Char;
                Base_Being_Used := GC.ParserInteger'VALUE
                    (Work_String(1..Work_String_Length));
I3              if (Base_Being_Used not in Valid_Base_Range) then
                    -- flag illegal bases as errors
                    LEM.Output_Message(
                        Current_Line
                      , Current_Column
                      , Work_String(1..Work_String_Length)
                      , LEM.Base_Out_Of_Legal_Range_Use_16);
                    Base_Being_Used := 16;
                    -- we use the maximum base to pass all the
                    -- extended_digits as legal.
                end if;

                Add_Next_Char_To_Source_Rep;  -- save the base delimiter
                Get_Next_Char;

C3              case Next_Char is
                    when 'A' .. 'F' | 'a' .. 'f' | Digit =>
                        Scan_Based_Integer(Base_Being_Used);
                    when ASCII.UNDERLINE =>     -- '_'
                        -- flag illegal leading under line
                        LEM.Output_Message(
                            Current_Line
                          , Current_Column
                          , LEM.Leading_Underline);
                        -- not flagging an integer consisting of a single
                        -- under line as a trailing radix point case.
                        Scan_Based_Integer(Base_Being_Used);
                    when '.' =>
```

```
                -- flag leading radix point as an error
                LEM.Output_Message(
                    Current_Line
                    , Current_Column
                    , LEM.Digit_Needed_Before_Radix_Point);
            when ASCII.SHARP |          -- '#'
                 ASCII.COLON =>         -- ':'
                -- flag missing field as an error
                LEM.Output_Message(
                    Current_Line
                    , Current_Column
                    , LEM.No_Integer_In_Based_Number);

                -- based_literal_delimiter_mismatch handled in
                -- next case statement.
            when others =>
                -- flag missing field as an error
                LEM.Output_Message(
                    Current_Line
                    , Current_Column
                    , LEM.No_Integer_In_Based_Number);
        end case;

C4      case Next_Char is
            when '.' =>
                Seen_Radix_Point := true;  -- have a real_literal
                Add_Next_Char_To_Source_Rep;

                Get_Next_Char;
C5              case Next_Char is
                    when 'A' .. 'F' | 'a' .. 'f' | Digit =>
                        Scan_Based_Integer(Base_Being_Used);
                        -- check and flag multiple radix points
W3                      while (Next_Char = '.') and then
                            ((Look_Ahead(1) in digit) or
                            (Look_Ahead(1) in 'A' .. 'F') or
                            (Look_Ahead(1) in 'a' .. 'f')) loop
                            LEM.Output_Message(
                                Current_Line
                                , Current_Column
                                , LEM.Too_Many_Radix_Points);
                            Add_Next_Char_To_Source_Rep;
                            Get_Next_Char;
                            Scan_Based_Integer(Base_Being_Used);
                        end loop;
                    when ASCII.UNDERLINE =>         -- '_'
                        -- flag illegal leading under lined
                        LEM.Output_Message(
                            Current_Line
                            , Current_Column
                            , LEM.Leading_Underline);
                        -- not flagging an integer consisting of
                        -- a single underline as a trailing
```

```
                    -- radix point case.
                    Scan_Based_Integer(Base_Being_Used);
                when others =>
                    -- flag trailing radix point as an error
                    LEM.Output_Message(
                        Current_Line
                        , Current_Column
                        , LEM.Digit_Needed_After_Radix_Point);
            end case;

C6          case Next_Char is
                when ASCII.SHARP |           -- '#'
                     ASCII.COLON =>          -- ':'

                    Add_Next_Char_To_Source_Rep;
                    -- save the base delimiter

I4                  if (Next_Char /= Based_Literal_Delimiter)
                    then
                        -- flag based_literal delimiter
                        -- mismatch as an error
                        LEM.Output_Message(
                            Current_Line
                            , Current_Column
                            ,   "Opener: "
                            & Based_Literal_Delimiter
                            & " Closer: " & Next_Char,
                        LEM.Based_Literal_Delimiter_Mismatch);
                    end if;

                    Get_Next_Char; -- after base delimiter
                    -- check for and process exponent
                    Scan_Exponent;

                when others =>
                    -- flag missing second
                    -- based_literal delimiter as an error
                    LEM.Output_Message(
                        Current_Line
                        , Current_Column,
                    LEM.Missing_Second_Based_Literal_Delimiter);
            end case;

        when ASCII.SHARP |           -- '#'
             ASCII.COLON =>          -- ':'
            -- have an integer_literal
            Seen_Radix_Point := false;
            -- save the base delimiter
            Add_Next_Char_To_Source_Rep;

I5          if (Next_Char /= Based_Literal_Delimiter) then
            -- flag based_literal delimiter mismatch error
                LEM.Output_Message(
```

```
                                          Current_Line
                                        , Current_Column
                                        ,   "Opener: "  & Based_Literal_Delimiter
                                          & " Closer: " & Next_Char
                                        , LEM.Based_Literal_Delimiter_Mismatch);
                                end if;

                                Get_Next_Char;  -- get character after base
                                                -- delimiter
                                Scan_Exponent;  -- check for and process exponent

                            when others =>
                                -- assume an integer_literal
                                Seen_Radix_Point := false;
                                -- flag missing second
                                -- based_literal delimiter as an error
                                LEM.Output_Message(
                                    Current_Line
                                    , Current_Column
                                    , LEM.Missing_Second_Based_Literal_Delimiter);
                        end case;

                    end if;

                --we have an integer_literal
                when others =>
                    Seen_Radix_Point := false;  -- have an integer_literal
                    Scan_Exponent;  -- check for and process exponent
            end case;

            -- one last error check
I6          if (Next_Char in Upper_Case_Letter) or
               (Next_Char in Lower_Case_Letter) then
                -- flag missing space between numeric_literal and
                -- identifier (including RW) as an error.
                LEM.Output_Message
                    ( Current_Line
                    , Current_Column
                    , LEM.Space_Must_Separate_Num_And_Ids);
            end if;

            -- now store the source representation of the token found.
            Set_CST_Source_Rep(Work_String(1..Work_String_Length));

        end Scan_Numeric_Literal;
```

This example has six case statements, six if statements, and three while statements. Each statement is calculated as its basic value plus its scope. No knots exist in this example to be counted.

Branches:

```
C1 = (2 * 3) + I1 + I2 = 76
C2 = (2 * 3) + W1 + W2 = 20
C3 = (2 * 5) = 10
C4 = (2 * 3) + C5 + C6 + I5 = 29
C5 = (2 * 3) + W3 = 13
C6 = (2 * 2) + I4 = 7

I1 = 4 + C2 = 24
I2 = 4 + C3 + C4 + I3 = 46
I3 = 3
I4 = 3
I5 = 3
I6 = 3

W1 = 7
W2 = 7
W3 = 7

MEBOW = 258
```

It is interesting to note that this MEBOW value is only slightly larger than the value for the much shorter example given in Figure 12. The reason is that this example shows well-structured code. This code has no explicit backwards branches, and no knots exist. This procedure's execution will always proceed from the top to the bottom, except for the implicit backwards branches because of the while statements.

The information flow calculation for this procedure is quite simple. All of the variables used other than the three locally-declared variables are global. If these variables are on the left side of an assignment, they are counted for fan-out. If they are used in any other location, they are counted for fan-in. One variable, "Work_String_Length," is used for both and is counted as both fan-out and fan-in.

Variables counted for fan-out:

```
CST.gram_sym_val
Work_String_Length
Seen_Radix_Point
```

Variables counted for fan-in:

```
PT.NumericTokenValue;
Next_Char
Look_Ahead
Digit
Current_Line
Current_Column
LEM.Too_Many_Radix_Points
LEM.Leading_Underline
LEM.Digit_Needed_After_Radix_Point
GC.ParserInteger'VALUE
Work_String
Work_String_Length
Valid_Base_Range
LEM.Base_Out_Of_Legal_Range_Use_16
LEM.Leading_Underline
LEM.Digit_Needed_Before_Radix_Point
LEM.No_Integer_In_Based_Number
LEM.Based_Literal_Delimiter_Mismatch
LEM.Missing_Second_Based_Literal_Delimiter
Upper_Case_Letter
Lower_Case_Letter
LEM.Space_Must_Separate_Num_And_Ids
```

INFO is defined in equation 4 as (fan-in * fan-out) ** 2. The number of variables counted for fan-in is 22. The number of variables counted for fan-out is three.

INFO = (22 * 3) ** 2 = 4356

Bibliography

Basili, Victor R. and Barry T. Perricone. "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, 27:1: 42-52 (January 1984).

Basili, Victor R. "Data Collection, Validation, and Analysis," Tutorial on Models and Metrics for Software Management and Engineering, edited by Victor R. Basili. New York: IEEE Computer Society Press, 1980.

Basili, Victor R. and Robert W. Reiter, Jr. "Evaluating Automatable Measures of Software Development," Tutorial on Models and Metrics for Software Management and Engineering, edited by Victor R. Basili. New York: IEEE Computer Society Press, 1980.

Boehm, B. W. and others. "Quantitative Evaluation of Software Quality," Tutorial on Models and Metrics for Software Management and Engineering, edited by Victor R. Basili. New York: IEEE Computer Society Press, 1980.

Conte, S.D. and others. Software Engineering Metrics and Models. Menlo Park CA: The Benjamin/Cummings Publishing Company, Inc, 1986.

Cote, V. and others, "Software Metrics: An Overview of Recent Results," The Journal of Systems and Software, 8: 121-131 (1988).

Curtis, Bill and others, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," Tutorial on Models and Metrics for Software Management and Engineering, edited by Victor R. Basili. Silver Spring MD: IEEE Computer Society Press, 1980.

Curtis, Bill and others, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," Tutorial on Models and Metrics for Software Management and Engineering, edited by Victor R. Basili. Silver Spring MD: IEEE Computer Society Press, 1980.

Department of the Air Force. Software Maintainability - Evaluation Guide. AFOTEC Pamphlet 800-2, Vol. 3. Albuquerque: HQ AFOTEC, 28 January 1988.

Hamer, Peter G. and Gillian D. Frewin, "M. H. Halstead's Software Science - A Critical Examination," Proceedings 6th International Conference on Software Engineering. 197-205. New York: IEEE Computer Society Press, 1982.

Hansen, Wilfred J., "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)," ACM SIGPLAN Notices, 13:3: 29-33 (March 1978).

Harrison, Warren and Curtis Cook, "A Micro/Macro Measure of Software Complexity," The Journal of Systems and Software, 7:3: 213-219 (September 1987).

Harrison, Warren and others, "Applying Software Complexity Metrics to Program Maintenance," IEEE Computer, 15:9: 65--79 (September 1982).

Henry, Sallie and Dennis Kafura, "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, SE-7:5: 510-518 (September 1981).

Henry, Sallie and others, "On the Relationships Among Three Software Metrics," Structured Testing, edited by Thomas J. McCabe. Silver Spring MD: IEEE Computer Society Press, 1983.

Howatt, Major James W., Professor, School of Engineering. Personal Correspondence. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 4 Nov 1988.

Howatt, Major James W., Professor, School of Engineering. Personal Correspondence. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 17 July 1988.

Jayaprakash, S. and others, "MEBOW: A Comprehensive Measure of Control Flow Complexity," Proceedings of the IEEE Computer Software and Applications Conference. 238-244. New York: IEEE Press, 1987.

Kafura, Dennis and James Canning, "A Validation of Software Metrics Using Many Metrics and Two Resources," Proceedings 8th International Conference on Software Engineering. 378-385. New York: IEEE Computer Society Press, 1985.

Kafura, Dennis and Geereddy R. Reddy, "The Use of Software Complexity Metrics in Software Maintenance," IEEE Transactions on Software Engineering, SE-13:3: 335-343 (March 1987).

Kearney, Joseph K. and others, "Software Complexity Measurement," Communications of the ACM, 29:11: 1044-1050 (November 1986).

Kitchenham, Barbara A., "An Evaluation of Software Structure Metrics," Proceedings of the IEEE Computer Software and Applications Conference: 369-376. Washington, D.C.: Computer Society Press of the IEEE, 1988.

Levitin, Anany V., "How to Measure Software Size, and How Not To," Proceedings of the IEEE Computer Software and Applications Conference. 314-318. New York: IEEE Press, 1986.

Li, H. F., and W. K. Cheung, "An Empirical Study of Software Metrics," IEEE Transactions on Software Engineering, SE-13:6: 697-708 (June 1987).

Lynn, Capt. Bernie. "Verification Study Results on the Software Maintainability Questionnaire (AFOTECP 800-2, Volume 3) and Evaluation Threshold (3.3)." HQ AFOTEC/LG5, Kirtland AFB, NM, 17 April 1985.

McCabe, Thomas J. "A Complexity Measure," Structured Testing, edited by Thomas J. McCabe. Silver Spring MD: IEEE Computer Society Press, 1983.

Myers, Glenford J., "An Extension to the Cyclomatic Measure of Program Complexity," ACM SIGPLAN Notices, 12:10: 61-64 (October 1977).

Oviedo, Enrique I., "Control Flow, Data Flow and Program Complexity," Proceedings of the IEEE Computer Software and Applications Conference: 146-152. New York: IEEE Press, 1980.

Page-Jones, Meilir. The Practical Guide to Structured Systems Design. New York: Yourdon Press, 1980.

Peercy, David E., "A Software Maintainability Evaluation Methodology," IEEE Transactions on Software Engineering, SE-7:4: 343-351 (July 1981).

Prather, Ronald E., "An Axiomatic Theory of Software Complexity Measure," The Computer Journal 27:4: 340-347 (1984).

Ramamurthy, Bina and Austin Melton, "A Synthesis of Software Science Metrics and the Cyclomatic Number," Proceedings of the IEEE Computer Software and Applications Conference. 308-313. New York: IEEE Press, 1986.

Rodriguez, Volney, and Wei-Tek Tsai, "Software Metrics Interpretation Through Experimentation," Proceedings of the IEEE Computer Software and Applications Conference: 368-374. New York: IEEE Press, 1986.

Schneidewind, Norman F., "The State of Software Maintenance," IEEE Transactions on Software Engineering, SE-13:3: 303-310 (March 1987).

Shen, Vincent Y, and others, "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support,", IEEE Transactions on Software Engineering, SE-9:2: 155-165 (March 1983).

Shepperd, Martin, "A Critique of Cyclomatic Complexity as a Software Metric," Software Engineering Journal: 30-36 (March 1988).

Walsh, Thomas J. "A Software Reliability Study Using a Complexity Measure." Structured Testing, edited by Thomas J. McCabe. Silver Spring MD: IEEE Computer Society Press, 1983.

Woodward, Martin R., "A Measure of Control Flow Complexity in Program Text," Structured Testing, edited by Thomas J. McCabe. Silver Spring MD: IEEE Computer Society Press, 1983.

VITA

Captain Stephen K. Johnson was born on 22 April 1962. His birth was recorded in Corvallis, Oregon, although he rarely admits that in public. After moving to California as a child, he graduated from Soquel High School in 1980, and accepted an appointment to the United States Air Force Academy. He received a Bachelor of Science degree in Computer Science and a regular commission into the Air Force.

Captain Johnson's first assignment was with the Strategic Air Command at Edwards AFB, California, in support of the B-1B Combined Test Force. He was responsible for measuring the maintainability of the B-1B's Defensive Avionics System's software and developed an appreciation for AFOTEC's maintainability evaluation guidelines. Captain Johnson entered the School of Engineering at the Air Force Institute of Technology in June 1987.

Permanent address: None

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
AFIT/GCS/ENG/88D-10

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
School of Engineering

6b. OFFICE SYMBOL (If applicable)
AFIT/ENG

7a. NAME OF MONITORING ORGANIZATION

6c. ADDRESS (City, State, and ZIP Code)
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583

7b. ADDRESS (City, State, and ZIP Code)

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
HQ AFOTEC

8b. OFFICE SYMBOL (If applicable)
LG5

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)
Kirtland AFB NM 87117-7001

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
See Box 19

12. PERSONAL AUTHOR(S)
Stephen K. Johnson, B.S., Capt, USAF

13a. TYPE OF REPORT
MS Thesis

13b. TIME COVERED
FROM ________ TO ______

14. DATE OF REPORT (Year, Month, Day)
1988 December

15. PAGE COUNT
122

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Maintainability, Measurement
Computer Programming

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: MODIFYING AFOTEC'S SOFTWARE MAINTAINABILITY EVALUATION GUIDELINES

Thesis Chairman: James W. Howatt, Major, USAF
Assistant Professor of Electrical and Computer Engineering

Approved for release in
Accordance with AFR 190-1
10 Jan 89

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
James W. Howatt, Major, USAF

22b. TELEPHONE (Include Area Code)
(513) 255-6913

22c. OFFICE SYMBOL
AFIT/ENG

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

The purpose of this study was to survey automatable software maintainability metrics for inclusion in the Air Force Operational Test and Evaluation Center's (AFOTEC'S) software maintainability evaluations. This research was looking for metrics that would measure maintainability, could be automated, and would fit into existing guidelines. First, a set of software complexity metrics was investigated. Then, a set of criteria to determine if a complexity metric measures maintainability was developed. After comparing the metrics to the criteria, a subset of two metrics that met the criteria better than any other metrics was derived

The software complexity metrics evaluated were placed into three categories: size metrics, structure metrics, and hybrid metrics. The structure metrics include both data structure and control structure metrics. The hybrid metrics include metrics blended from two of the other groups, such as a combination of size and structure metrics.

The metric selection criteria included three categories: general applicability criteria, control flow complexity criteria, and data flow complexity criteria. An assumption was made that the metric or combination of metrics that met the most of these criteria would best reflect software maintainability. A combination of a data structure metric, information flow, and a control structure metric, MEasurement Based on Weights (MEBOW), was determined to meet more criteria than any other metric or combination of metrics. This hybrid metric was suggested for AFOTEC use.

Further information explaining theoretical and empirical justification for the use of these metrics was given. A description of techniques to determine metric threshold values was discussed, along with a procedure for metric validation. Finally, a theme of the limitations inherent in measuring maintainability with automatic metrics was elaborated.